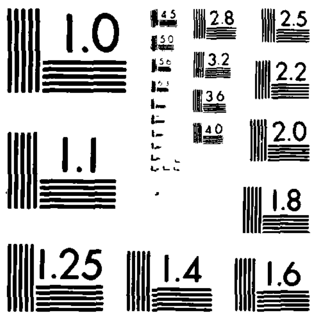
AD-A109 976 TEXAS INSTRUMENTS INC LEWISVILLE EQUIPMENT GROUP F/G 9/2
ADA INTEGRATED ENVIRONMENT III COMPUTER PROGRAM DEVELOPMENT SPE--ETC(U)
DEC 81 F30602-80-C-0293
UNCLASSIFIED RADC-TR-81-360-VOL-1 NL

1 of 2
AD
A109976


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

PHOTOGRAPH THIS SHEET

AD A109976

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

Texas Instruments Inc.,
Lewisville, TX Equipment Group - ACSL
ADA Integrated Environment III Computer
Program Development Specification. Interim Rept.
15 Sep. 80 - 15 Mar 81
Dec. 81
Contract F30602-80-C-0293 RADC-TR-81-360, Vol. I

DOCUMENT IDENTIFICATION

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR		
NTIS	GRA&I	☒
DTIC	TAB	☐
UNANNOUNCED		☐
JUSTIFICATION		
BY		
DISTRIBUTION /		
AVAILABILITY CODES		
DIST	AVAIL AND/OR SPECIAL	
A		

DISTRIBUTION STAMP

DTIC
ELECTE
S JAN 25 1982 D
D

DATE ACCESSIONED

DTIC
COPY
INSPECTED
3

82 01 12 010

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DTIC FORM OCT 79 70A

DOCUMENT PROCESSING SHEET

RADC-TR-81-360, Vol I (of four)
Interim Report
December 1981

ADA INTEGRATED ENVIRONMENT III COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

Texas Instruments, Inc.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

RADC-TR-81-360, Volume I (of four) has been reviewed and is approved for publication.

APPROVED: Elizabeth S. Kean
ELIZABETH S. KEAN
Project Engineer

APPROVED: [illegible]
JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss
JOHN P. HUSS
Acting Chief, Plans Office

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-360, Vol I (of four)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ADA INTEGRATED ENVIRONMENT III COMPUTER PROGRAM DEVELOPMENT SPECIFICATION		5. TYPE OF REPORT & PERIOD COVERED Interim Report 15 Sep 80 - 15 Mar 81
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0293
9. PERFORMING ORGANIZATION NAME AND ADDRESS Texas Instruments Incorporated Equipment Group-ACSL, P O Box 405, M.S. 3407 Lewisville TX 75067		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62204F/33126F/62702F 55811919
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COES) Griffiss AFB NY 13441		12. REPORT DATE December 1981
		13. NUMBER OF PAGES 171
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Elizabeth S. Kean (COES)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada	MAPSE	AIE
Compiler	Kernel	Integrated environment
Database	Debugger	Editor
KAPSE	APSE	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Ada Integrated Environment (AIE) consists of a set of software tools intended to support design, development and maintenance of embedded computer software. A significant portion of an AIE includes software systems and tools residing and executing on a host computer (or set of computers). This set is known as an Ada Programming Support Environment (APSE). This B-5 Specification describes, in detail, the design for a minimal APSE, called a MAPSE. The MAPSE is the foundation upon which an

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

APSE is built and will provide comprehensive support throughout the design, development and maintenance of Ada software. The MAPSE tools described in this specification include an Ada compiler, linker/loader, debugger, editor, and configuration management tools. The kernel (KAPSE) will provide the interfaces (user, host, tool), database support, and facilities for executing Ada programs (runtime support system).

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

TABLE of CONTENTS

LIST of FIGURES

LIST of EXAMPLES

SECTION 1

SCOPE

1.1 Identification

This part of this specification establishes the requirements for performance, design, test, and qualification of a computer program identified as the Ada Software Environment (ASE). This CPCI provides a machine-independent interface between any Ada program (including tools within the Ada Integrated Environment) and the computer system on which it executes.

1.2 Functional Summary

The Ada Software Environment provides a virtual interface between the user of the Ada Integrated Environment (AIE) and the particular host system upon which the AIE is installed. This interface is designed to be machine and operating system independent; in effect, it defines a transportable Ada virtual machine whose features are available on all actual host machines. The Ada Software Environment CPCI is an implementation of a Kernel Ada Programming Support Environment (KAPSE) and the KAPSE virtual interface [RADC80].

The Ada Software Environment virtual interface has three aspects:

1. Intra-Program -- The run-time environment is provided to implement the semantics of the Ada programming language. Typical areas are subprogram linkage, storage management, tasking, and high-level input/output.

2. Inter-Program -- One Ada program may invoke another and communicate with it using a high-level I/O, which is implemented through a low-level inter-program communication package. Service requests are made to the AIE database by sending messages to the program in which it is implemented.

3. User -- The user communicates with the AIE through a terminal whose actual characteristics are mapped by the environment into a standard form. The Ada Software Environment also provides the command language with which a user requests services and the interpreter that translates these requests into invocations of appropriate programs.

SECTION 2

APPLICABLE DOCUMENTS

2.1 Program Definition Documents

[DoD80A]	Requirements for Ada Programming Support Environments: "STONEMAN", DoD (February 1980).
[RADC80]	Revised Statement of Work for Ada Integrated Environments, RADC, Griffiss Air Force Base, NY (March 1980).
[SOFT80A]	Ada Compiler Validation Capability: Long Range Plan, SofTech Inc., Waltham, MA (February 1980).
[SOFT80B]	Draft Ada Compiler Validation Implementers' Guide, SofTech Inc., Waltham, MA (October 1980).

2.2 *Military Specifications and Standards*

[DoD80B]	Reference Manual for the Ada Programming Language: Proposed Standard Document, DoD (July 1980) (reprinted November 1980).
[DoD80C]	Airborne Computer Instruction Set Architecture MIL-STD-1750A, DoD (March 1980).

SECTION 3

REQUIREMENTS

3.1 Introduction

This section presents the requirements for the Ada Software Environment (ASE) component of the Ada Integrated Environment. This component implements a KAPSE (Kernel Ada Programming Support Environment) for the Ada Integrated Environment (AIE). The ASE provides a machine-independent interface between any Ada program (including tools within the Ada Integrated Environment) and the computer system on which it executes. This interface is designed to be machine and operating system independent; in effect, it defines a transportable Ada virtual machine whose features are available on all actual host machines. The ASE is described in terms of the following components:

1. Ada Execution Environment.
2. Storage Management.
3. Task Management.
4. Program Management.
5. APSE Manager.
6. Input/Output.

3.1.1 Program Interfaces

The KAPSE virtual interface that is implemented by the ASE can be viewed from several aspects. From the viewpoint of an Ada program that is executing on a host system (Figure 3-1), ASE defines the interface through which access is given to the host machine and its operating system. Access to the host machine is made through a collection of run-time support routines that implement the semantics of the Ada language; an Ada program must be bound with these routines to produce an executable entity. Access to the host operating system is through routines that define a transportable virtual interface; if these routines can implemented under a host operating system, the ASE can be installed. On an embedded system (Figure 3-2), the ASE becomes an operating system kernel that provides services such as clock management that are normally associated with a host operating system.

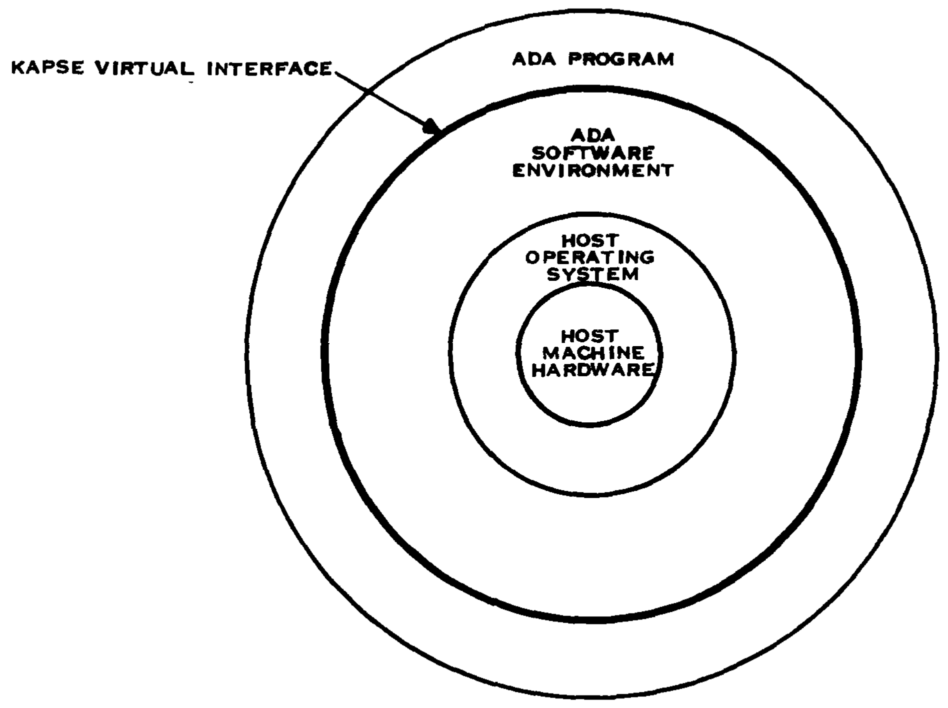


Figure 3-1 KAPSE Virtual Interface on a Host Machine

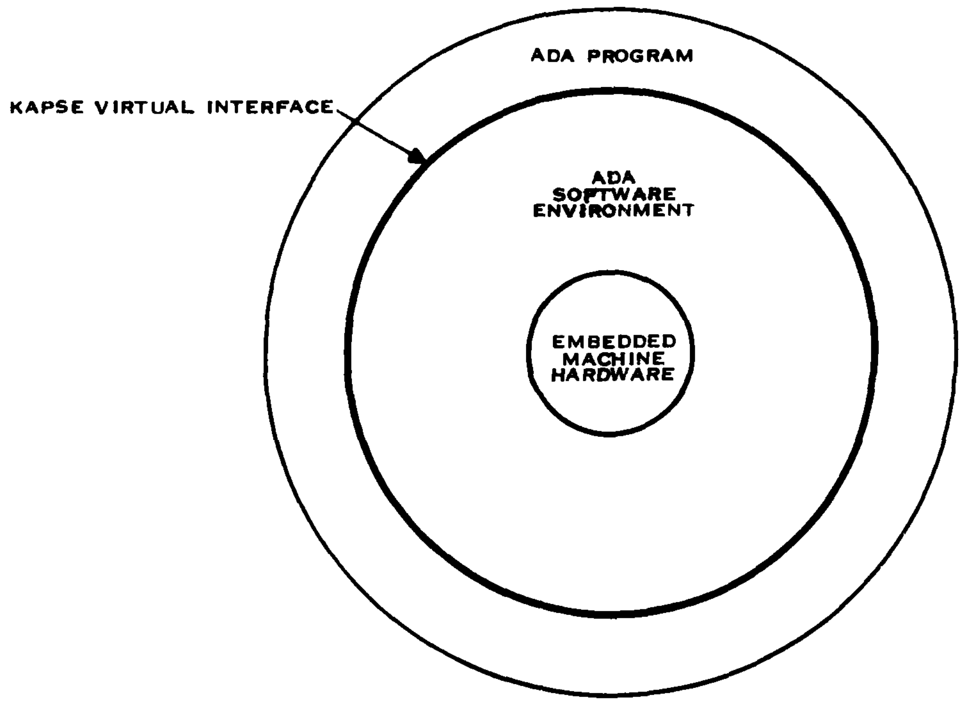


Figure 3-2 KAPSE Virtual Interface on a Target Machine

Figure 3-3 shows the relationship of the ASE to a host operating system under which several AIE users are executing concurrently. For each user of the AIE, there is an instance of the executive program, the ASE component that interfaces with the user and permits invocation and control of Ada programs. While there may be more than one program executing under the control of a user, each is an independent entity with respect to resource management and scheduling by the host operating system. The APSE manager is a collection of AIE programs that must be available to manage global resources such as the database or to communicate with remote machines. These programs are logically associated with a user who serves as the AIE operator, although there is no requirement for an operator or terminal be dedicated to this function. If sharing of code among programs is supported by the host operating system, the object code for the executive program and the ASE run-time support components will be configured so only one copy need be kept in memory.

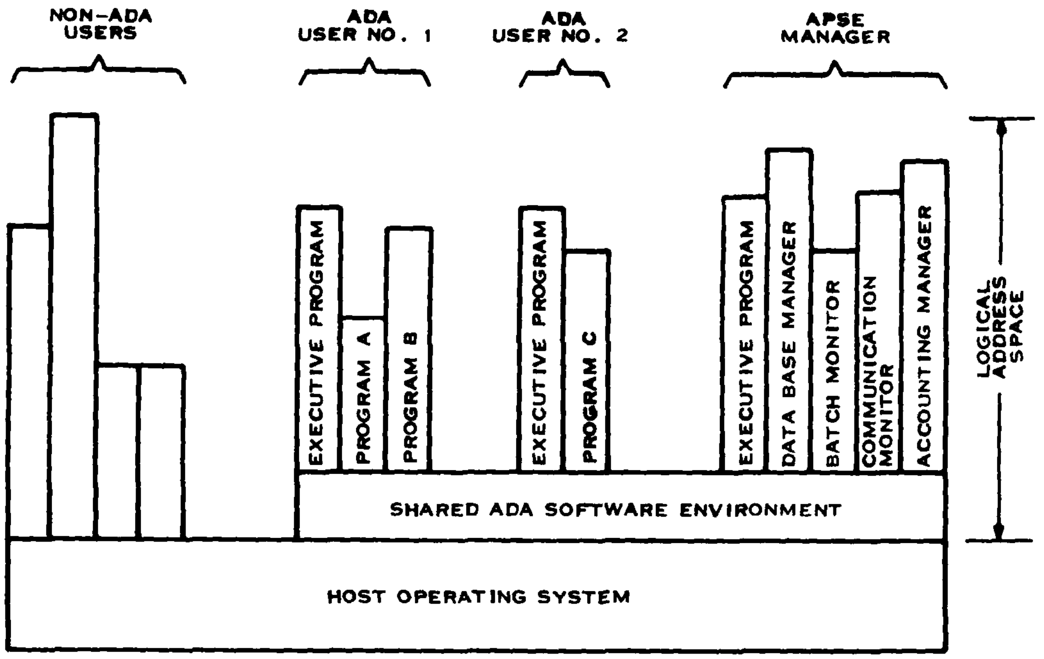


Figure 3-3 ASE Interface to a Host Operating System

When executing under VM/370 (Figure 3-4), the host machine is configured into multiple virtual machines, each of which appears to have the full power of the 370 architecture. The simulation of virtual machines on a single actual machine is performed by the VM/370 Control Program (CP). Each virtual machine can run its own operating system, which may be single user (e.g., CMS, the conversational monitor system) or multi-user (e.g., MVS, the general-purpose multi-tasking operating system). In this environment, each AIE user is given an individual virtual machine for which the ASE serves as a multi-programming, virtual-memory operating system; that is, the ASE supports concurrent execution of programs, each of which has its own virtual address space.

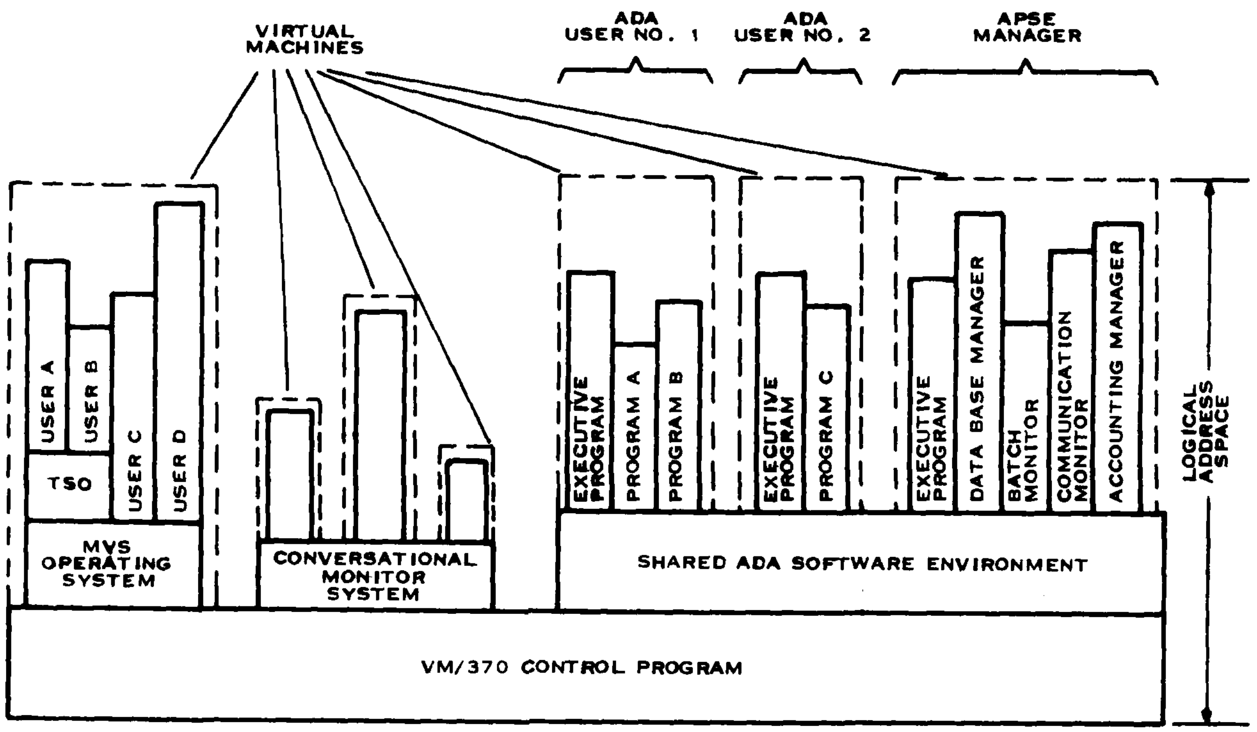


Figure 3-4 ASE Interface to VM/370

Figure 3-5 shows the ASE from the viewpoint of a user at his terminal. The terminal is connected to the ASE executive program through a device controller task that is customized for particular types of terminals. From the terminal the user can instantiate the command language interpreter task (CLI), which translates command language into invocations of programs. Commands are provided with which to request services from the APSE manager.

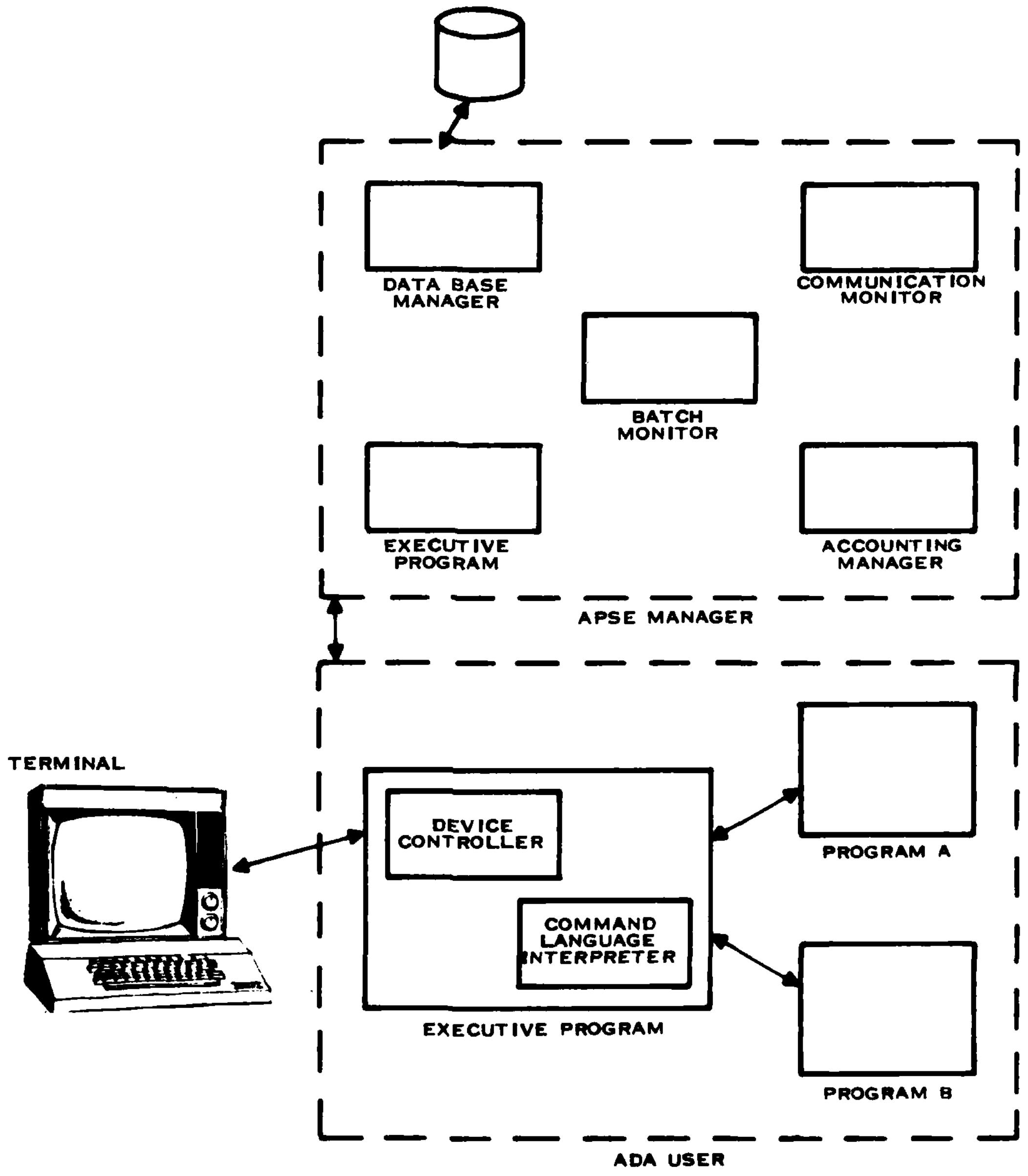


Figure 3-5 ASE Interface to a User

The following subsections discuss the interfaces that exist between the ASE and other components of the Ada Integrated Environment. Although all tools depend on ASE to provide the environment for their execution, only those components are listed for which there is an interface requiring coordinated implementation with the ASE.

3.1.1.1 Ada Optimizing Compiler

The design of the execution environment, storage management, and task management components of the ASE form a specification for the translation of certain features of Ada. This design must be coordinated with the design of the code generation phase of the Ada optimizing compiler.

3.1.1.2 Program Binder

To load a program and place in execution, the ASE must access data prepared by the binder. The binder either builds the subprogram reference tables that are used by the subprogram linkage handlers or prepares auxiliary information from which the tables are built by the loader function of the ASE. Global packages are elaborated and program parameters are passed based upon tables prepared by the binder.

3.1.1.3 Program Debugger

The debugger controls a program by sending messages to the KAPSE interface task, the component of the ASE that resides in the address space of every Ada program and provides access to external programs via the KAPSE virtual interface. These messages permit the debugger to examine and modify the state of a program. For the debugger to be essentially portable, the debugger/ASE interface must be designed so the debugger can get needed data while minimizing its knowledge of the underlying hardware. In particular, requests should be expressed in terms of the Ada abstract machine, not architectural features of the target machine. The debugger can load, start, and suspend a program as well as examine its state and data structures.

3.1.1.4 Host Operating System

The KAPSE virtual interface implemented by the ASE is the gateway through which an AIE user requests service from the host operating system. This interface must be designed so it can be provided in an identical form on a wide class of hosts. The following points discuss the host operating system features that are essential to the implementation of the ASE; the Perkin-Elmer 8/32 [PER78] under OS/32 [PER79] and IBM 370 [IBM76] under VM/370 [IBM79B] are used as examples. (The IBM Virtual Machine/System Product (VM/SP) [IBM80] has been introduced to extend the capabilities of the VM/370 Control Program (CP), the component of VM/370 that provides virtual machines. The ASE will be designed to take advantage of VM/SP features if that program product is installed on a host system.)

1. Time services -- The ASE must be able to access the current date and time-of-day to implement the package CALENDAR; the DELAY statement, timed entry calls, and DELAY alternatives on SELECT statements require that a program schedule a timer interrupt at

either a specific time in the future or periodically.

a. OS/32
Service call 2, code 8 returns time-of-day; service call 2, code 9 returns the date. Timer interrupts can be scheduled using service call 2, code 23.

b. VM/370
The 370 store clock (STCK) instruction returns the value of the time-of-day clock that is maintained by the (real) 370 hardware. A clock interrupt can be generated at a specific time by using the set clock comparator (SCKC) instruction to store a value into the comparator associated with the time-of-day clock.

2. Semaphores -- The implementation of Ada tasking described in the section on task management is based on the use of semaphores as a low-level synchronization primitive. A semaphore permits a task to request (a P operation) and release (a V operation) exclusive access to a data structure or critical section of code. If access cannot be granted immediately, the requesting task must wait until the resource is released by some other task. Implementation of the semaphore abstraction requires an even lower level of synchronization primitive to ensure that a given semaphore operation can be completed as an indivisible operation. Since this lowest level of synchronization can result in a scheduling decision, it is closely related to the manipulation of the task scheduling data structures, which in general requires that the program be placed in a state that momentarily blocks external stimuli. It is important to recognize that many semaphore operations will not require a scheduling decision and optimize the implementation accordingly.

a. OS/32
The ideal way to implement semaphores would be to use the user-defined service call (SVC 14) to context switch to a state in which all external program stimuli (e.g., program-level timer interrupts, inter-program messages) are masked. Since this service call requires processing by OS/32, it is desirable to treat as a special case the situation in which a scheduling decision is not required. Two flags are placed in a task management structure and are manipulated by the test-and-set (TS) instruction. (The test-and-set instruction tests a bit and sets it to "1" as an indivisible operation; the bit changing from "0" to "1" is used to indicate that a task has gained exclusive access to a resource.)

In general, each semaphore operation begins with an attempt to set the first flag. If it does not change state or if it does but examination of the semaphore indicates a scheduling decision is required, a SVC 14 call is made to relinquish the processor to the task scheduler component of task management. The scheduler executes with a program status word that masks all (program-level) interrupts while the remainder of the semaphore operation is performed. If the task does execute the semaphore operation without requiring a scheduling decision, the first flag is reset to release access to all semaphores.

A single flag is not sufficient since an external stimulus may cause a second task to be activated that attempts a semaphore operation before the first task releases access to all semaphores. In this case, the scheduler must suspend the second task; a second flag is used to signal the first task that a scheduling decision is pending. Each semaphore operation concludes with a test-and-set operation applied to the second flag; if the flag changes state, the task makes a SVC 14 call to activate the scheduler. Since the only way for a task to be interrupted while performing a semaphore operation is for an external stimulus to occur, most semaphore operations will not require use of SVC 14.

b. VM/370
The same approach will be used with the test-and-set and service call instructions of the 370.

3. Asynchronous entry from keyboard -- There must be some form of "break" from the keyboard with which the user can interrupt execution of a program and give control to the ASE executive.

a. OS/32
The device controller will issue a read request to the keyboard with "proceed" option. The program receives a trap when the request completes due to a user entering a full buffer, a carriage return, or a "break" key. The trap will be routed to the device controller as a simulated interrupt.

b. VM/370
The device controller task of the ASE executive program will respond to the keyboard interrupts and examine the keys that have been depressed to look for a "break."

4. Asynchronous inter-program messages -- Inter-program messages are used to send data and control signals from one program to another. Message arrival must cause a simulated interrupt to ensure that waiting tasks are reactivated and that one program can exert positive control over another.

a. OS/32
Two methods for transferring data between programs are available; both cause a trap within the receiving program. The primary method is to use service call 6 to send a message from one program to another (in increments of 64 bytes or less). In the special case where one program is communicating with a second (e.g., the database manager)

that is in turn performing file input / output, it is possible for the second program to verify the access rights of the first program and use the "send logical unit" function of service call 6 to give the first program direct connection to the file. The first program can perform I/O operations directly, but only after the second program has enforced security checks.

b. VM/370
Inter-program and inter-machine communication will be performed using the Virtual Machine Communication Facility (VMCF) of VM/370 [JEN79, IBM79B], which causes a simulated hardware external interrupt. Under VM/SP, the Inter-User Communication Vehicle (IUCV) [IBM80] will be used since it supports communication with the control program as well as other virtual machines.

5. Dynamic program invocation -- One program must be able to place another in execution and receive the identity of the new program so messages can be passed to it. Ada programs are loaded in two steps: a canonical program image is loaded which has within it code that can access the AIE database and load the particular program that is being invoked. The canonical program can have any format that facilitates host system loading.

a. OS/32
Supervisor call 6 supports the dynamic loading of a program. The name of the program (for purposes of inter-program communication) is assigned when the request to load is made; an option is available with which to specify the amount of data (i.e., heap) space to be given to the program. Parameters can be passed to the new program so it can establish communication with its creator. (The ASE will be organized to place code that must be available to every Ada program in OS/32 library segments so that code can be shared among all AIE users.)

b. VM/370
Loading under VM/370 has two aspects: loading the virtual machine in which a user executes and loading programs within that machine. When a user enters AIE, he logs in to VM/370 and begins execution in a "saved" system (virtual machine) that contains within it the minimal amount of code that suffices to attach saved discontiguous segments (using the DIAGNOSE instruction, code X'60') containing the ASE code that must always be available for a program to execute. (This ASE code is not placed in the saved system since modifications would require all users to replace their virtual machines; modifications needed be made only in the saved segments.) The code for the canonical ASE program is contained within the saved segments that are shared by all AIE virtual machines. When a request is made to the ASE to invoke a program, a separate virtual data space is allocated

for the canonical program, which is added to the program scheduling data structures that are maintained by the ASE. When the canonical program becomes active, it communicates with the database to load the specified program.

6. Host file system -- All (disk) file operations will be performed through the database manager program of the APSE manager. On a host machine, the database will typically use host system files to maintain its directories and to store database objects. Sequential and random access to data within files are required.

   a. OS/32
      OS/32 has a file system that stores objects by volume name, file name, extension, and file class. The file name consists of one to eight alphanumeric characters, the first of which must be alphabetic; the extension is zero to three alphanumeric characters. Thus, there are enough names under which to store database objects. An indexed file structure is supported that permits data to be accessed either sequentially or randomly and provides for dynamic expansion of files.

   b. VM/370
      VM/370 does not support files directly. Simulated disks are available but may not be safely shared in a read/write mode. All disk space will be managed by the database manager with data being transferred to users via inter-program communication. The file management code will implemented using low-level disk handlers based on those in CMS.

A transportable user interface to host operating systems is described in [HAL80]; the four host system requirements [SCH79] that are most likely to cause problems and their relation to the ASE requirements are:

1. Character sets -- All terminal I/O goes through a device-specific instance of the device controller, so translation to a standard character set is possible.

2. Passing arguments to running programs -- This feature is implemented using inter-program communication.

3. Random access to files -- Both sequential and random access to files is assumed.

4. Execution of subtasks -- In the ASE, this feature is provided via the weaker requirement to be able to load the canonical program. One program controls another through messages that cause software interrupts in the subordinate program's KAPSE interface task.)

3.1.1.5 User

The ASE interface to the user is through his terminal, using either the command language of the executive control task or the command interpreter.

3.1.2 Functional Descriptions

3.1.2.1 Ada Execution Environment

The Ada execution environment is the component of ASE that specifies the implementation of the Ada abstract machine on a particular processor. This specification is used by the code generation phase of the Ada optimizing compiler to translate Ada language features into machine instructions. For features whose semantics are too complex to be translated inline, the code generator will emit calls to out-of-line routines supplied in this component.

3.1.2.2 Storage Management

The storage management package contains routines to allocate and deallocate objects whose lifetimes are dynamic. Such objects are either ACCESS objects that are manipulated directly by the user via the allocator NEW (and possibly by the deallocator UNSAFE_DEALLOCATION) or are created indirectly by the ASE in response to user statements (e.g., task elaboration requires allocation of data structures by the task management package.)

3.1.2.3 Task Management

This component of the ASE implements the tasking features of Ada. Included are task elaboration and termination, rendezvous, and interrupt handling.

3.1.2.4 Program Management

The Ada Integrated Environment requires that the ASE support the execution of one or more programs by a user; this component provides the interface between the user and the programs he invokes. Included are the executive control language with which the user controls the configuration of his terminal, the command language with which programs are invoked interactively, and the program invocation package which supports the invocation of one program by another.

3.1.2.5 APSE Manager

The APSE manager is a collection of programs that are associated with the Ada Integrated Environment instead of a particular user. When the AIE is brought up on a machine, the APSE manager is created as an anonymous user who has the same structure (i.e., an executive program is controlling other programs) as other users but is not associated with a particular terminal. The APSE manager contains programs such as the database manager and batch monitor that provide a system-wide service.

3.1.2.6 Input/Output

The input/output component of ASE provides both the low-level I/O routines with which data transfers are made and the high-level routines that are oriented to the needs of an Ada programmer. Conversion between internal (binary) and external (ASCII) representations can be specified using templates.

3.2 Detailed Functional Requirements

3.2.1 Ada Execution Environment

The Ada language supports features such as exceptions and packages whose semantics are complex enough that inline translation will not be appropriate on most target machines. For such features, the Ada compiler will conserve program space by generating linkages to "out-of-line" routines that are collected within the Ada execution environment, the ASE component that provides a run-time interface between the target machine and the abstract machine that models the semantics of Ada. This section describes this interface by identifying the data structures that are required by the Ada abstract machine and the language constructs that will typically profit from out-of-line translation.

3.2.1.1 Inputs

This component of ASE is collection of out-of-line code sequences, each of which has its own inputs. In a functional sense, the inputs to this component are service requests made at the level of the Ada abstract machine.

3.2.1.2 Processing

The content of this component depends upon the target machine for which the ASE is being prepared: fewer run-time support routines will be required if the target machine is similar to the Ada abstract machine. This subsection describes certain data structures of the Ada abstract machine and enumerates language features whose implementation is complex enough that out-of-line code may be advantageous. Subsequent subsections will discuss features such as tasking that involve substantial processing that is not apparent at the language level.

3.2.1.2.1 Subprogram Structure

The object module that the Ada optimizing compiler generates for a subprogram consists of a code section and a constant section as depicted in Figure 3-6. The code section contains the machine instructions into which the subprogram was translated. In general, this code will require references to constant data that cannot be translated as "immediate" instruction operands. Examples are long string literals and structured objects with attribute CONSTANT. Such constant data are grouped into a constant section that is logically disjoint from the code section. (This distinction is important because constants and code have different properties in certain environments: passing the address of a constant places constraints on calls between parallel overlays; the military standard 1750A architecture [DoD80C] uses different mapping registers to access instructions and data.)

The entry point of a subprogram is the first location in its code section. The last portion of a code section contains what is called the epilogue of the subprogram, which is the code that must be executed to exit the subprogram. In addition to a call to a subprogram exit handler, this code contains any special processing required to deallocated compiler-generated structures such as subheaps. The displacement to this code is stored at a dedicated location in the constant section so an exception that is propagated through this subprogram can force execution of the epilogue. The size of the static portion of the subprogram stack frame is also stored at a dedicated location in the constant section and is used by the call handler for stack frame allocation.

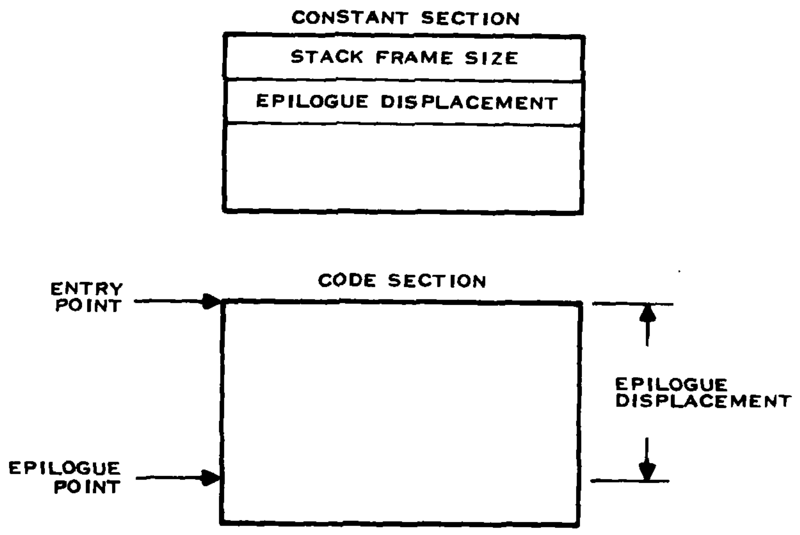


Figure 3-6 Subprogram Structure

3.2.1.2.2 Program Segmentation

The section on modular software in [TI81A] shows the advantages of partitioning a program into segments. The sharing of code among programs is facilitated if inter-segment references are decoupled so each segment is independent of the addresses at which subprograms are allocated in other segments. Segmentation is performed by the Program Binder component of the Ada Programming Toolset. The binder collects one or more subprograms into a segment with the structure shown in Figure 3-7. The interrelationships among the segments comprising a program are made through a structure called a segment table that is associated with the program. An entry in the segment table contains the addresses of the four components of the associated segment. The code and constant sections are concatenations of the code and constant sections, respectively, of the constituent subprograms. The dictionaries for code and constant sections are used to locate individual sections within the concatenation. Each subprogram section is accessed indirectly through the index of an entry in the dictionary of the segment in which the reference occurs.

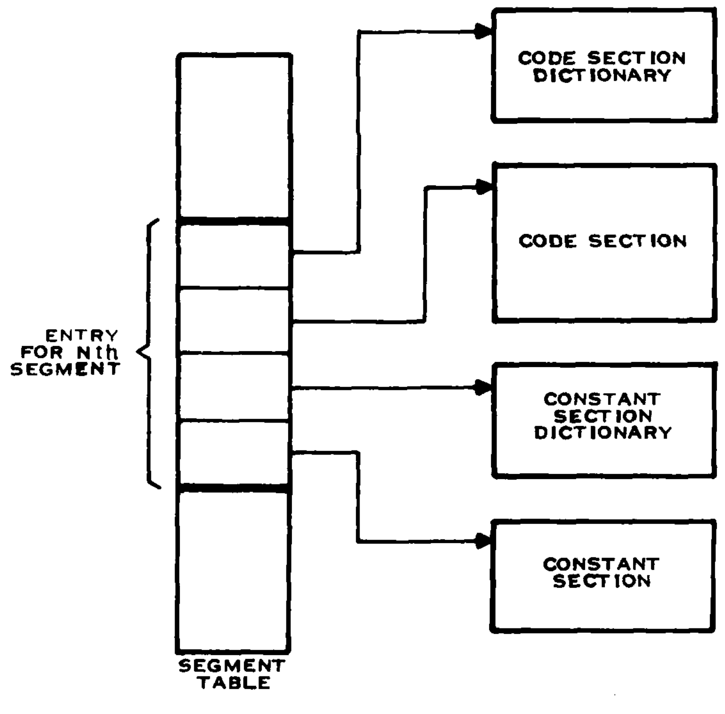


Figure 3-7 Segment Organization

Consider the code section and dictionary depicted in Figure 3-8. If a call to subprogram 2 is made in subprogram 1, the reference to subprogram 2 is made in terms of its index "2" in the segment code section dictionary. The second entry in the dictionary has a "0" flag bit, which means the associated subprogram is local to the current segment and the remainder of the dictionary entry contains the displacement of the subprogram entry point from the beginning of the segment code section. External references (e.g., the subprogram with index 5) are expressed in terms of the number of the segment in which the subprogram is bound and the index of that subprogram's entry of in the associated dictionary. (The subprogram entry in the external dictionary will normally be internal; it will be external if the subprogram has been replaced by a "patched" version [TI81A].) The segment constant section dictionary has the same structure as the code section dictionary except it contains only internal references, each of which is a displacement relative to the beginning of the segment constant section. A reference to the constant section of another subprogram is made through a run-time support routine whose input is the index of the subprogram in the code section dictionary. That index is mapped through code dictionaries until an internal entry is found for the subprogram, at which point the corresponding entry in the associated constant section dictionary is used to find the base address of the constant section. Items within the constant section are accessed using displacements relative to the beginning of the constant section.

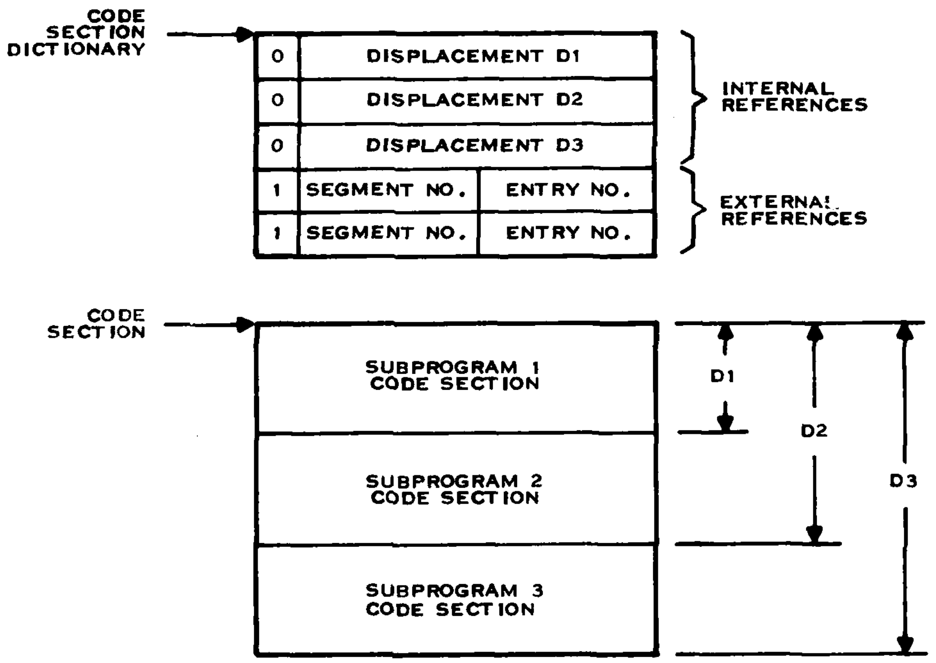


Figure 3-8 Code Section Dictionary

If each entry in a dictionary is sixteen bits, of which seven are used for the segment number and eight for the entry index, this segmentation scheme supports 128 segments, each of which contains as many as 256 subprogram. On machines such as the 370 and 8/32 that require halfword alignment, the fifteen bit displacement can be interpreted as a halfword displacement, so each section can be as large as 65536 bytes. Thus, the collection of code (or constant) sections can potentially fill 2**23 bytes, which is one half the logical address space of both the 370 and the 8/32.

3.2.1.2.3 Subprogram Linkage

The visibility rules of the Ada language cause subprogram linkage to be much more complex than the "branch and link" instructions that are typically provided on target machines. Subprogram entry and exit require significant modifications to the data structures used to model the addressing environment of the Ada abstract machine and to save subprogram context. The typical approach to the consolidation of linkage code is to use "branch and link" instructions to transfer control first from one subprogram to another and then to an out-of-line routine that updates the Ada abstract machine. As is discussed in detail in the modular software section of [TI81A], a more versatile approach is to invoke an out-of-line call handler from the context of the calling subprogram and have the handler modify the machine state before transferring to the called subprogram. (One advantage of this technique is that automatic overlay loading and memory mapping changes can be supported since the call handler executes before the called subprogram is entered and thus has an opportunity to perform special processing.)

Since modular software is likely to have many subprogram calls, it is important to use an efficient instruction sequence at the point of call. For machines such as the 370 and 8/32 that have an adequate number of registers, the address of the call handler will be held in a dedicated register. In this case, a call requires four bytes of code space: a two-byte "branch and link register" instruction followed by the two-byte index of the called subprogram in the active code section dictionary. Since the linkage routine for subprogram exit is called only once per subprogram, it will be invoked by a four-byte branch instruction to a fixed displacement relative to the call handler.

Special linkage handlers will be supplied for subprograms that use a non-standard linkage (the pragma INTERFACE).

3.2.1.2.4 Stack Management

Ada is a block structured language for which it is desirable to generate object code that can be shared among tasks within a program (and among programs within an APSE multi-programming system). This leads to the use of a stack (last-in, first-out) discipline for the allocation of storage for objects declared in a block, subprogram, package, or task. Each task in a program is given a stack region from which its objects are allocated upon entry to a block, subprogram, package, or task entry. For each such entity, allocation occurs in two stages since the size of some objects cannot be determined prior to their elaboration. A single allocation is made for the storage associated with all objects whose sizes were known at compile time. (This allocation is made by the call handler based upon the stack frame size parameter in the constant section of the unit.) Each object with computed size is allocated by extending the stack and storing the address of the created space in a descriptor (contained within the first allocation area) that represents the object. (On machines such as the 370 that have a limited displacement field in data reference instructions, "long" structures will be treated as if they were dynamic so the static portion of the stack frame can be covered by one base register.) The routines that administer the stack region will be translated out-of-line since shared code can be used to check for and raise the STORAGE_ERROR exception.

A display [GRI71] will be used to provide addressability to stack frames of nested of subprograms. The display will be implemented as an array of addresses, each of which points to the stack frame of a subprogram that is in scope. Storage for the display will be contained within the task control block of each task.

Figure 3-9 shows the three areas into which a stack frame is partitioned. The areas for static and dynamic objects were discussed above. The linkage data area is used by the various linkage handlers to save and restore the following subprogram context data:

1. Register Set -- The caller's register set is saved by a called subprogram in the linkage area of the caller's stack frame. (For machines such as the 370 and 8/32 that have separate floating point registers, these registers are not saved in the linkage area; it would be wasteful to always allocate space for these registers when many subprograms will not use them. If a subprogram references a floating point register, the code generator will save that register at subprogram entry and restore it at exit.)

2. Return Address -- One representation of the return address is typically saved in one of the caller's registers; depending on the specific form of segmentation that is being used (e.g., automatic overlay loading), it may be necessary to save the return address as a segment, displacement pair.

3. Old frame Pointer -- The caller's frame pointer is saved in the callee's linkage area so the return context (linkage area) can be found at subprogram exit.

4. Exception Handler -- This field contains the displacement in the constant section of the descriptor for the active exception handler of the associated subprogram. At subprogram entry, this field is initialized to zero, which indicates that no exception handler has been established.

5. Display Pointer -- If a subprogram has subprograms declared within it, the display most be updated so the subprograms can access variables of their parent. When the subprogram is entered, it saves the display pointer at its lexical level in its linkage area and moves its frame pointer into the display. At exit, the program restores the display pointer from the linkage area.

6. Dependent Task List -- This field, which is initialized to NULL, is used by the task management package to link together the task control blocks of all tasks that are dependent on the task associated with this stack region. A subprogram cannot exit if this field is not NULL since this condition indicates that dependent tasks remain active and can access the frame of the subprogram.

The caller's top-of-stack pointer is not saved since it is equal to the callees frame pointer.)

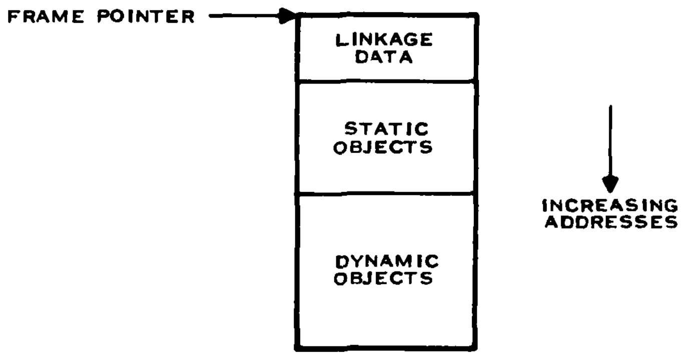


Figure 3-9 Stack Frame Layout

Objects declared within a block are allocated storage within the stack frame of the unit in which the block is declared. Storage for static objects is merged into the static area of the unit and allocated when that unit is entered. Dynamic objects are allocated at entry to the block and deallocated at exit. (Storage for static objects may be shared by blocks that are not nested.)

3.2.1.2.5 Parameter Passing and Function Results

Parameters are passed by having the caller build a list of actual parameters in its stack frame and pass the address of that list to the callee in a standard register; the argument list is treated as part of the local storage of the callee. For a formal parameter of scalar or access type that has mode IN or IN OUT, a copy of the corresponding actual is placed in the argument list; upon return, the caller copies each formal parameter of mode OUT or IN OUT back into the actual. For a parameter of array, record, or private type, the address of the actual parameter is placed in the argument list, and the callee uses that address as a pointer through which to access the actual.

Having the caller build an parameter list within its stack frame (instead of pushing parameters onto a stack) permits the compiler to treat each parameter list as a distinct structure that is subject to optimization. If a subprogram has only IN parameters and for a particular call all the actuals are known at compile time, the parameter list for that call can be allocated in the constant section of the caller. If a call occurs within a loop and some of its IN paramaters do not change within that loop, part of the initialization of the parameter list can be promoted out of the loop.

A function result of scalar or access type is returned in a register (or registers) of the caller. (On machines such as the 370 and 8/32 that have special floating point registers, a REAL result is returned in such a register.) For other types, the result is returned in three steps: the result is copied to the beginning of the callee's stack frame, the stack frame of the caller is expanded to include the result, and the address of the result is returned is a register. A routine will be provided with which the compiler can restore the stack frame when the result the result is no longer needed. (It is possible to copy the result while executing in the callee's register set since the caller's registers are saved in the caller's frame and are not overwritten during the copy.)

3.2.1.2.6 Register Usage

For the 370 and 8/32, dedicated registers are assigned to hold the following items. For all but the last item, the corresponding register is initialized by the call handler at subprogram entry.

1. Code Section Address -- This address points to the entry point of the active subprogram and is used as a base register for program-relative branch instructions.

2. Constant Section Address -- This address points to the beginning of the constant section of the active subprogram and is used as a base register to access constants (that are not "immediate.")

3. Frame Pointer -- This value is used to access objects within the local frame of the active subprogram.

4. Task Control Block Address -- A task is represented in the ASE by a data structure called a task control block. In addition to data required to implement tasking, this structure also contains addresses that are used in subprogram linkage and variable access. In particular, the first field in the task control block is a "branch" instruction to the call handler that is being used by the task; thus, a "branch and link register" instruction that specifies the task control block as its destination causes a branch through the task control block to the entry handler. The task control block also contains the address of a vector of addresses through which other linkage handlers are accessed and the display, a vector of pointers into the stack that correspond to stack frames that are in scope.

5. Parameter List Address -- This address is used to access the list of actual arguments of the current subprogram.

6. Callee's Parameter List Address -- When a subprogram is called, the address of the callee's list of actual parameters is passed by the caller in a register. This register is loaded by the caller and used to build the list of actual parameters; the call handler transfers its content to the "parameter list address" register of the called subprogram.

Code generation for the 370 is complicated by the fact that the largest offset that is available within an instruction is 4095 bytes. (The 8/32 has an address mode with a 24-bit displacement field.) It is conceivable that one base register may not suffice to cover the code section, constant section, or stack frame of a subprogram. For the code section, the Ada compiler will examine the characteristics of the routine to determine if it is more efficient to allocate additional base registers or to modify the code section register as execution flows through the module. For the constant section and stack frame, the compiler will arrange items to place shorter ones at the beginning of an area and will use indirection to access long structures. (Within the constant section, any indirection will be through pointers that contain displacements relative to the section; this preserves position independence.)

3.2.1.2.7 Packages

Packages provide grouping of related entities; in the most general form, a package can contain declarations of types, data, and subprograms. Data declared within the package specification have the same lifetime as the scope in which the package is elaborated. The methods by which such data are allocated and accessed depend upon the relationship of the package declaration to other units.

If a package is declared within another unit, the package is represented by a descriptor within the stack frame of the unit. The package body is compiled as a procedure with a special exit sequence that does not deallocate the stack frame of the package body; instead, the stack frame of the subprogram in which the package is elaborated is extended to include the package frame and the address of the package frame is returned to be stored in the package descriptor. A subprogram that is declared within such a package has the address of the package frame passed as an implicit parameter that is used as a pseudo display pointer.

If a package is not declared within another unit, the package has the same lifetime as any program in which it is used. The analogue of the technique used to reference nested packages would be to create a dummy global stack frame that contains pointers to the storage of all global packages and to pass the appropriate pointer as an implicit parameter of each subprogram of a given package. However, a library unit can be compiled with knowledge of only those packages that it references directly. A consistent assignment of package pointers to the dummy frame would require that displacements not be specified until a program is bound; moreover sharing code among programs would require a consistent assignment for all programs referencing a particular package. This situation is the data analogue of code sharing and is resolved with a similar form of segmentation. Specifically, special entries are appended to the code section dictionary for use in referencing global packages. Each entry is a 16-bit index into a package table, an array containing the address of the storage area for each global package. Since there can be only a single instance of a global package in a program, there is no need to pass the package data pointer as an implicit pointer to each subprogram of such a package. Instead, each subprogram begins with a call to a routine that maps the code section dictionary index of the associated package into the address of that package's data area. The storage for global packages is allocated and initialized using the same technique as used by nested packages; however, the address of the storage is placed in the package table instead of being returned to the subprogram in which elaboration occurs. (The KAPSE interface task causes the task associated with the main program to begin execution with the elaboration of global packages, so storage for global packages is allocated within the stack region of this task.)

3.2.1.2.8 Tasks

Ada supports tasking at a high level of abstraction that requires a substantial run-time support package to manipulate complex data structures. Creation of a task object results in the allocation of a stack region in which task data are allocated and a control structure that is used to schedule the task for execution and to save its context when it is not executing. These structures must be reclaimed at task termination, an event that must be synchronized with the termination of dependent tasks. Through calls to task entries and execution of SELECT and ACCEPT statements, a process called rendezvous occurs that involves inter-task communication and synchronization. Tasks are described in detail in the section on task management.

3.2.1.2.9 Interrupts

A user can handle an interrupt within the Ada language by specifying that each occurrence of the interrupt be treated as a hardware-invoked call to a task entry. The interrupt entry is viewed as being called from a task with higher priority than any user-defined task; as a consequence, a rendezvous that is associated with an interrupt entry will preempt any user-defined task that is executing. Interface code must be provided to accept an interrupt, save the context of the interrupted task, accept any IN parameters from the hardware associated with the interrupt, restore the context of the interrupt handler, and place it in execution. The interface for interrupts is described in the section on task management.

3.2.1.2.10 Exceptions

Exception processing will generally require out-of-line code to examine subprogram linkage data to locate the first exception handler that can respond to a particular exception. This is an iterative process that must be able to make significant modifications to the state of a task to cause transfer of control across block boundaries.

Each exception handler for a block is represented by a descriptor stored in the constant section of the subprogram in which the block occurs. This descriptor contains two 16-bit items: the displacement of the exception handler code from the beginning of the code section of the subprogram and the displacement of the descriptor of the enclosing block (or 0 if there is none) from the beginning of the constant section. The descriptors are chained together so exceptions can be propagated.

A RAISE statement is translated into a call to one of:

```
procedure RAISE_EXCEPTION;
procedure RAISE_EXCEPTION( "exception representation" );
```

depending on whether a specific exception is specified; an exception is represented by the character string form of its identifier. (It would be possible to have the binder find all exceptions that are used in a program and assign an integer representation to each; however, such an approach might prevent sharing of code among programs since it might not be possible to select a consistent representation.)

When an exception occurs or a RAISE statement is executed, the exception code is stored in the control block of the associated task, and the exception handler field of the linkage area of the active subprogram is examined. If this field is not zero, it contains the displacement in the constant section of the descriptor for the active exception handler. The two fields of the descriptor are used to invoke the handler: the displacement of the enclosing handler is stored in the linkage area (so subsequent exceptions will be handled in that block) and the displacement of the handler is used to branch to it. If the exception handler field is zero, no handler has been specified, and the exception must be progagated to the subprogram that called the one under examination. The return address field in the linkage area of the current subprogram modified to point to a RAISE statement (with no parameter), and a branch is made to the epilogue point of the current subprogram. As a consequence, the exception is raised in the context of the caller.

Figure 3-10 shows the translation of a block with an exception handler. The statements of the block begin with a call to SET_EXCEPTION_HANDLER, which makes the exception handler active by storing the displacement of its descriptor in the linkage area. RESTORE_EXCEPTION_HANDLER terminates the block by using the link field of the active descriptor to restore the previous handler. The code for the exception handler begins at << LABEL_1 >>; if control reaches that point, an encoding of the exception has been stored in a register that is represented by the pseudo variable EXCEPTION_CODE. An IF statement is used to check against each of the exceptions recognized by this handler; if the exception is not handled, it is propagated by a RAISE statement. (A block with an OTHERS choice is translated similarly except the final ELSE clause of the IF statement contains the statments associated with the choice OTHERS instead of the RAISE statement.)

```
               begin
                 -- Statements.
               exception
                 when ERROR_1 | ERROR_2 =>
                   -- Handler A.
                 when ERROR_3 =>
                   -- Handler B.
               end


               begin
                 SET_EXCEPTION_HANDLER( "displacement
                   to exception descriptor" );

                 -- Statements.

                 RESTORE_EXCEPTION_HANDLER;
               goto << LABEL_2 >>;
<< LABEL_1 >>  null;
               -- Assume EXCEPTION_CODE has been
               --   loaded.
               if EXCEPTION_CODE = ERROR_1 or else
                  EXCEPTION_CODE = ERROR_2 then
                 -- Handler A.
               elsif EXCEPTION_CODE = ERROR_3 then
                 -- Handler B.
               else
                 raise;
               end if;
<< LABEL_2 >>  null;
```

Figure 3-10 Exception Handler

3.2.1.2.11 CASE Statement

The CASE statement requires the selection of one of several alternatives. If the range of alternatives is large or sparse, the CASE statement will be translated as a series of IF statements (that require no out-of-line code). Otherwise, an indirect branch will be made through a table in the constant section that contains the code section displacement of each alternative; out-of-line code can be used to check that the CASE expression is within the range covered by the table (some alternatives may be specified by OTHERS and not be given in the table) and to make the branch.

3.2.1.2.12 Access Objects

The allocator NEW permits the user to control (at execution time) the creation of objects; such an object is referenced through variables of access type and has a lifetime that lasts at least as long as the object is referenced by some name. Storage for an access object can be associated with its access type (via the STORAGE_SIZE length specification) or the scope in which the type is declared. Routines must be provided to initialize and reclaim the storage associated with an access type or scope, implement the allocator NEW, and define the generic procedure UNCHECKED_DEALLOCATION. The implementation of access objects is described in the section on storage management.

3.2.1.2.13 Data Type Support

It is likely that some of the Ada primitive data types will not be supported on a given target machine; for example, some machines do not provide string or floating point instructions. For such machines a collection of out-of-line routines will have be supplied to simulate the missing instructions In some cases, the routines can be in the form of an Ada package; in others, they can be most efficiently invoked using a special linkage whose design has been coordinated with the code generation model being employed by the Ada optimizing compiler.

A collection of overloaded functions will be provided that implement the attributes IMAGE and VALUE for scalars and subtypes.

3.2.1.3 Outputs

The outputs of this component are services performed to implement selected features of the Ada abstract machine in terms of the instruction set of the target machine.

3.2.1.4 Special Requirements

The implementation of this component is closely related to the design of the code generation phase of the Ada optimizing compiler. The routines included in this component and their calling sequences are highly dependent upon the architecture of the target machine.

3.2.2 Storage Management

The Ada language supports the creation of objects whose lifetimes are dynamic: an object comes into existence when the allocator NEW is executed within the body of a subprogram, not when a declarative part is elaborated. For such objects to be created, objects of access type must be declared to hold the access values returned by NEW. The objects designated by a particular access type form a collection for which the user can specify certain parameters. The length specification STORAGE_SIZE may be used to indicate how much storage should be made available for objects of the collection. The pragma CONTROLLED may be used to indicate the circumstance under which storage for access objects should be reclaimed. (An access object must remain allocated as long as it is still accessible.) The default reclamation strategy is garbage collection, in which storage is (potentially) reclaimed as soon as it becomes inaccessible; this is generally very expensive to implement since the storage management package must be able to locate all variables of access type and to determine all objects designated (directly or indirectly) by them. A much less expensive strategy, which is indicated by the pragma CONTROLLED, is to deallocate the storage for a collection at exit from the scope in which the associated access type is declared, since at that point there are no objects left to designate objects in the collection. In addition, the generic procedure UNCHECKED_DEALLOCATION may be used to notify the storage manager that a specific access object may be reclaimed.

3.2.2.1 Inputs

Figure 3-11 shows the declaration for the low-level storage management package, which consists of the package STORAGE_MANAGER and the generic procedure UNCHECKED_DEALLOCATION. This package manages a pool of storage called a heap that is partitioned into regions called subheaps. Each subheap contains one or more collections that have the same lifetime; this grouping makes it possible to reclaim collections instead of individual access objects. Each subheap is represented by an object of the limited private type SUBHEAP.

```
package STORAGE_MANAGER is

  subtype ADDRESS_RANGE is INTEGER range 0 .. MEMORY_SIZE;

  type SUBHEAP is limited private;

  procedure INITIALIZE_SUBHEAP( S          : out SUBHEAP;
                                SIZE       : ADDRESS_RANGE := 0;
                                EXPANDABLE : BOOLEAN := TRUE );

  procedure INITIALIZE_SUBHEAP( S          : out SUBHEAP;
                                SIBLING    : SUBHEAP;
                                SIZE       : ADDRESS_RANGE := 0;
                                EXPANDABLE : BOOLEAN := TRUE );

  procedure RECLAIM_SUBHEAP( S        : in out SUBHEAP;
                             SIBLINGS : BOOLEAN );

  procedure UNCHECKED_ALLOCATION( X    : out ADDRESS_RANGE;
                                  S    : SUBHEAP;
                                  SIZE : ADDRESS_RANGE );

  procedure UNCHECKED_ALLOCATION( X         : out ADDRESS_RANGE;
                                  S         : SUBHEAP;
                                  SIZE      : ADDRESS_RANGE;
                                  ALIGNMENT : ADDRESS_RANGE;
                                  OFFSET    : ADDRESS_RANGE );

procedure UNCHECKED_DEALLOCATION( X : in out ADDRESS_RANGE );

end STORAGE_MANAGER;

generic
  type OBJECT is limited private;
  type NAME is access OBJECT;
procedure UNCHECKED_DEALLOCATION( X : in out NAME );
```

Figure 3-11 Storage Management Package

3.2.2.2 Processing

When a program begins execution, it is given a pool of memory called a heap from which objects of various types can be allocated. The heap is managed by the package STORAGE_MANAGER, which is used by both application programs and the ASE. A user typically requests storage management service by a call to the allocator NEW, which is translated by the compiler into a call to subprogram UNCHECKED_ALLOCATION. The portions of the ASE that are bound with a program to provide its run-time environment will make extensive use of the storage manager to allocate internal data structures. For example, the task management package must allocate a task control block and stack region for each Ada task that it creates. Some calls may have to be made directly to the allocator UNCHECKED_ALLOCATION to allocate certain meta-level objects of the Ada abstract machine (such as the stack region) whose structure cannot be determined (completely) until execution time.

There are several advantages to partitioning the storage space into subheaps. The primary one is that it is possible to reclaim access objects at *scope exit* with a single call, without having to trace through linked data structures. Use of subheaps also permits run-time expansion of the heap (via service calls to the host operating system to acquire more space) and tends to localize the area from which storage is being allocated, thus decreasing fragmentation of the heap.

The algorithm used to manage storage within a subheap can be varied to suit the characteristics of a particular application. The default version supplied with the ASE is based upon the heap management package that was used in [TI81C]. A subheap is partitioned into a collection of packets, each of which is either allocated or deallocated. The first word of each packet is a descriptor with the definition given in Figure 3-12. The PACKET_SIZE field contains the total size of the packet in words (not storage units) so the least significant bit of the descriptor word can be used to indicate if the packet is allocated. (On a machine for which the storage unit is a word, packets would have to be allocated an even number of words to free the flag bit.) If a packet is in use, the access object that it contains begins in the word following the descriptor. If the address of a packet (or an access object) is given, it is possible to determine the availability of a packet for allocation, its size, and the address of its successor. With this information, it is possible to search a subheap (given the address of its first packet) to allocate a new access object; as the subheap is searched, contiguous deallocated packets can be consolidated. Deallocating an object only requires that the ALLOCATED bit of its descriptor be reset; in particular, a packet need not contain information identifying the subheap of which it is a member.

```
WORD     : constant 4; -- Storage unit is byte, 4 bytes per word.
LAST_BIT : constant STORAGE_UNIT * WORD - 1;

type PACKET_DESCRIPTOR is
  record
    PACKET_SIZE : integer range 0 .. MEMORY_SIZE / WORD;
    ALLOCATED   : BOOLEAN;
  end record;

for PACKET_DESCRIPTOR use
  record
    PACKET_SIZE at 0 range 0 .. LAST_BIT - 1;
    ALLOCATED   at 0 range LAST_BIT .. LAST_BIT;
  end record;
```

Figure 3-12 Heap Packet Descriptor

The following sections describe the subprograms that provide user-level interface to the storage management package. This version has been optimized for execution speed by not providing garbage collection, the reclamation of storage for an object as soon as it is no longer designated *by some variable. A weaker form a reclamation* is implemented, reclamation at exit from the scope in which a access type is declared. A subsequent section discusses the trade-offs involved in providing garbage collection.

3.2.2.2.1 Package Body STORAGE_MANAGER

Execution begins in the storage management package with the elaboration of the body for the package STORAGE_MANAGER; since this package is a library unit, its elaboration occurs as part of the initialization of a program, before control is given to the body of the main program. This body must initialize the internal data structures used to manage the heap. In particular, it must link together the (possibly discontiguous) areas of storage that are to form the heap.

3.2.2.2.2 Subprogram INITIALIZE_SUBHEAP

This subprogram creates and initializes the data structures that form a subheap of the program heap. The compiler must generate a call to this subprogram in the prologue of any subprogram or block containing the declaration of an access type. The first call in a prologue must be to the version that does not have the parameter SIBLING; subsequent calls must be to the other version. The SIBLING argument is used to link together all the subheaps in a scope so they may be deallocated with one call to RECLAIM_SUBHEAP. The parameter SIZE is used to specify the amount of storage to be allocated to a collection, as may be done with the length specification STORAGE_SIZE; a value of 0 causes a default size to be selected. If space in a subheap is exhausted and the subheap was initialized with parameter EXPANDABLE set to TRUE, the memory manager will attempt to expand the subheap by acquiring more space from the program heap.

3.2.2.2.3 Subprogram RECLAIM_SUBHEAP

This subprogram reclaims the space within a subheap and returns it to the program heap for re-use. If the parameter SIBLINGS is TRUE, all other subheaps that were declared in the same scope are also reclaimed. Since a subheap is basically an object in the program heap, reclamation requires only that the subheap data structures be deallocated within the context of the program heap.

3.2.2.2.4 Subprogram UNCHECKED_ALLOCATION

This subprogram is invoked by the compiler to implement the allocator NEW. It returns the address of an object in the specified subheap with (at least) the requested size. An overloaded version of this subprogram is provided for use when an access object requires alignment to begin at OFFSET address units beyond the boundary specified by ALIGNMENT. (Since the compiler generates calls to this subprogram, it need not be generic.)

The particular algorithm that is used for allocation can be tailored to the characteristics of a program. The default version searches each region of the subheap, from low addresses to high, for a packet with the requested amount of space; adjacent unallocated packets are consolidated. If a packet is found that is large enough, it is trimmed to the requested size with unneeded space remaining unallocated. If a packet is not found and the subheap is expandable, an additional region of the subheap is requested from the program heap; if it cannot be allocated, the STORAGE_ERROR exeception is raised.

3.2.2.2.5 Subprogram UNCHECKED_DEALLOCATION

The user may signal the storage manager that a particular access object may be reclaimed by instantiating the generic subprogram UNCHECK_DEALLOCATION (Figure 3-13) and calling the subprogram with the associated access variable as a parameter. The generic subprogram uses the ADDRESS attribute to pass the address of the access object to the version of UNCHECKED_DEALLOCATION in the storage management package, which performs the actual deallocation. For the default version of the package, the only required processing is to set the flag ALLOCATED to FALSE. The IN OUT parameter X is set to NULL so it cannot be used to reference the deallocated object.

```
with STORAGE_MANAGER;
generic
  type OBJECT is limited private;
  type NAME is access OBJECT;
procedure UNCHECKED_DEALLOCATION( X : in out NAME ) is
  use STORAGE_MANAGER;
  TEMP: ADDRESS_RANGE;
begin
  TEMP := X.all'ADDRESS;
  STORAGE_MANAGER.UNCHECKED_DEALLOCATION( TEMP );
  X := null;
end UNCHECKED_DEALLOCATION;
```

Figure 3-13 Library Unit UNCHECKED_DEALLOCATION

3.2.2.3 Outputs

The outputs are discussed in the description of the individual subprograms in the package STORAGE_MANAGER.

3.2.2.4 Special Requirements

3.2.2.4.1 Deallocation Following Exceptions

Successful reclamation of subheaps requires that the compiler insert a call to RECLAIM_SUBHEAP in the "epilogue" of each subprogram in which an access type is declared. This call must be executed even in the presence of an exception. In particular, suppose subprogam A calls subprogram B which calls subprogram C, B declares an access type, and an exception is raised in C that can be handled A but not B or C. For collection, associated with B's access type to be reclaimed, the implemention of exception handling cannot permit execution to resume in A without giving B a chance to perform its epilogue processing.

3.2.2.4.2 Initialization

The package STORAGE_MANAGER is not free to use all of the capabilities of access types within in its own code since that code implements some of those capabilities. In particular, the compiler must not emit calls to INITIALIZE_SUBHEAP within the package body since they would violate the rules given in [DoD80B, 10.5] for elaboration of library units. Since the compiler cannot allocate a subheap descriptor, the package cannot call the allocator NEW; in fact, a lower level subprogram than UNCHECKED_ALLOCATION will have to be used to avoid deadlock with respect to heap access.

3.2.2.4.3 Synchronized Access to the Heap

The package STORAGE_MANAGER must provide synchronized access to the heap data structures since its procedures may be called simultaneously from several tasks. One approach is to use Ada tasking to provide the synchronization. This may not be acceptable since it may lead to another circular dependence: task creation causes compiler-generated calls to the storage manager that cannot be suppressed. It may be preferable to use directly the same low-level synchroniztion mechanism (semaphores) that is used within task management. Although such an approach would be more difficult to implement, it would permit synchronization to be performed efficiently on a subheap basis.

3.2.2.4.4 Garbage Collection

Garbage collection is a method of reclaiming space in a heap by marking all objects that are accessible (directly or indirectly) from an application program and returning to an unallocated state all those that are not marked (and thus unaccessible). In a functional sense, garbage collection is preferable to deallocation at scope exit since many embedded system applications will have access types that are program or task global, so will never be reclaimed. The subprogram UNCHECKED_DEALLOCATION provides precise control over deallocation, but it is subject to misuse. Since garbage collection must examine all of the storage associated with an access type, it typically requires a significant amount of execution time to complete. For embedded system applications, waiting to perform garbage collection until storage is exhausted leads to unacceptable periods during which algorithmic processing is suspended. [STE75], [WAD76], and [BAK78] (among others) have investigated garbage collection algorithms that are suited for real time and/or multiprocessing environments.

Implemention of a concurrent garbage collection algorithm such as [WAD76] requires two forms of assistance from the compiler. The foremost is the development of data structures that are available at run-time with which to locate all references to objects in the heap. Such references may occur within objects in the heap, stack, or registers. They may involve structures with variant parts, unconstrained array types, or shared usage of the same stack location (or register) within a subprogram or block. Special code may have to be emitted by the compiler to mark lifetimes of access variables. The second form of assistance from compiler is required whenever an access value is assigned. Since the garbage collector is marking accessible objects in parallel with the application tasks, it is possible that the variable into which the access value is assigned has already been examined by the collector. To ensure the associated access object is not overlooked, the compiler must emit special code to mark the object.

It appears that implementation of concurrent garbage collection will require that the compiler perform a significant amount of additional processing and will impose a serious space and time overhead at execution time. We will investigate the details of such an implementation and access its practicality for embedded systems. For garbage collection to be feasible, it will be implemented in such a manner that there is no run-time penalty for applications in which it is not used.

3.2.3 Task Management

The task management package provides a collection of routines that implement tasking, the Ada feature that supports multiple concurrent sites of execution within a single program. Tasking is provided in Ada at a high enough level of abstraction that its implementation is not apparent from the language semantics; moreover that implementation involves complex time-dependent interactions that are subject to subtle errors and gross inefficiencies. This section presents a design for the translation of Ada tasking statements and for the implementation of the run-time support routines that comprise this package; most algorithms are described in terms of Ada pseudo-code that treats in detail problems of synchronization. This design has been influenced by [EVA80], [EVE80], [HAB80], and [TI81C].

3.2.3.1 Inputs

This section presents the inputs to the task management package in terms of a functional description of the data structures associated with tasking. After the structures have been discussed, an example of their manipulation during rendezvous is given. Figure 3-14 gives a (partial) Ada declaration for the data structures that represent a task entry and a task control block; they are the fundamental structures manipulated by the task management package.

```
type NONNEGATIVE is INTEGER range 0 .. INTEGER'LAST;

type ADDRESS_RANGE is INTEGER range 0 .. MEMORY_SIZE;

type ENTRY_RANGE is INTEGER range -1 .. INTEGER'LAST;

type ENTRY_DESCRIPTOR;

type ENTRY_DESCRIPTOR_ADDRESS is
  access ENTRY_DESCRIPTOR;

type TASK_CONTROL_BLOCK(
       MAX_ENTRY_NUMBER : ENTRY_RANGE );

type TASK_CONTROL_BLOCK_ADDRESS is
  access TASK_CONTROL_BLOCK;

type ENTRY_DESCRIPTOR is
  record
    OPEN_ENTRY         : ENTRY_DESCRIPTOR_ADDRESS;
    WAITING_TASK_LIST  : TASK_CONTROL_BLOCK_ADDRESS;
    CODE_ADDRESS       : ADDRESS_RANGE;
  end record;

type ENTRY_DESCRIPTOR_ARRAY is
  array( INTEGER range <> ) of ENTRY_DESCRIPTOR_ADDRESS;

type ENTRY_DESCRIPTOR_ARRAY_ADDRESS is
  access ENTRY_DESCRIPTOR_ARRAY;

type TASK_CONTROL_BLOCK(
       MAX_ENTRY_NUMBER : ENTRY_RANGE) is
  record -- (partial definition)
    MUTEX            : SEMAPHORE;
    SUSPEND          : SEMAPHORE;
    PARTNER_LIST     : TASK_CONTROL_BLOCK_ADDRESS;
    QUEUE_LINK       : TASK_CONTROL_BLOCK_ADDRESS;
    OPEN_ENTRY_LIST  : ENTRY_DESCRIPTOR_ADDRESS;
    SELECTED_ENTRY   : ENTRY_DESCRIPTOR_ADDRESS;
    ARG_LIST         : ADDRESS_RANGE;
    SAVED_PRIORITY   : PRIORITY;
    DELAY_TIME       : DURATION;
    ENTRY_TIME       : DURATION;
    ENTRY_ARRAY      : array( 0 .. MAX_ENTRY_NUMBER )
                         of ENTRY_DESCRIPTOR_ARRAY_ADDRESS;
  end record;
```

Figure 3-14 Task Data Structures

3.2.3.1.1 Task Control Block

Each task in an Ada program is represented within the ASE by its task control block. This structure contains two classes of data. One class is parameters of the Ada abstract machine that are needed within application code; examples are the display and addresses of linkage handlers. The second class of data is used by the task management package itself; examples are the task priority, fields through which task control blocks are linked, mutual exclusion and task suspension semaphores, and the task context (machine state) when it is not executing. (For convenience the task control block is described as a single structure; on systems that support memory mapping, it will be split into two structures so the part used for task management can be placed in the address space that is associated with the executive.) The task control block is so fundamental that the linkage protocol for a target machine will be chosen to provide efficient access to it: a register will generally be dedicated to point to the task control block. The following fields within the task control block will be used in the discussion of task management algorithms.

1. MUTEX -- This semaphore is used to provide exclusive access to data structures associated with the task. In particular, it is used to control access to the list of all open entries (OPEN_ENTRY_LIST) and the queue of tasks that are waiting for an entry call to be accepted (WAITING_TASK_LIST).

2. SUSPEND -- The task associated with this control block uses this semaphore to suspend itself to await some change of state. For example, if the task calls an entry that is not open, the processor is relinquished via a semaphore P operation applied to SUSPEND; when another task executes an ACCEPT statement on the entry and completes rendezvous, the suspended task is reactivated by a V operation.

3. PARTNER_LIST -- This field is used to link together the task control blocks of all tasks that have made entry calls to this task and are now in rendezvous. The list is last-in, first-out (linked via QUEUE_LINK) with the partner of the inner-most rendezvous at the head of the list.

4. QUEUE_LINK -- This field is used to link this task control block into queues such as PARTNER_LIST and WAITING_TASK_LIST.

5. OPEN_ENTRY_LIST -- This field points to the list of entry descriptors of "open" entries -- entries for which an ACCEPT statement has been executed and rendezvous is awaited. The list is circularly linked through field OPEN_ENTRY of the entry descriptors; OPEN_ENTRY_LIST points to the last entry that was inserted. (This method for linking has the property that an entry descriptor is in the list if and only if its OPEN_ENTRY field is not NULL.)

6. SELECTED_ENTRY -- When an entry call occurs after an ACCEPT statement for the entry, the calling task sets the SELECTED_ENTRY

field of the called task's control block to indicate which entry was selected for rendezvous; if the ACCEPT occurred within a SELECT statement, several entries may have been open. This field is also used by a task making a timed entry call; it is set to the entry descriptor address of the entry upon which the call is made.

7. ARG_LIST -- When an entry call is made, the address of the argument list passed to the entry is stored in the ARG_LIST field of the calling task's control block. When rendezvous occurs, the entry retrieves its arguments by calling function ENTRY_ARGUMENT_LIST, which searches the PARTNER_LIST of the called task to find the control block of the caller and return its ARG_LIST pointer.

8. SAVED_PRIORITY -- The Ada rules for task priority during rendezvous [DoD80B, 9.8] require that the priority of the called task be at least as great as that of the calling task. Since ACCEPT statements can be nested, the priority of the called task may have to be adjusted an arbitrary number of times. This adjustment is performed by saving the current priority of the called task in the SAVED_PRIORITY field of the control block of the calling task at the point the calling task is accepted for rendezvous and is entered into the PARTNER_LIST of the called task. The priority is restored when the rendezvous is completed and the calling task is removed.

9. DELAY_TIME -- This field is set to indicate to the clock manager the time at which reactivation from a DELAY statement is desired. The field is set to 0.0 by the clock manager if the delay expires and by the task if reactivation is no longer desired.

10. ENTRY TIME -- This field has the same function as DELAY_TIME except it refers to a timed entry call instead of a DELAY statement.

11. ENTRY_ARRAY -- This array contains an item for entry that is declared in the task; each item is a pointer to an array of entry descriptors. If an entry is a member of a family, the array has the same index range as the family, and there is one descriptor per member. An entry that is not a member of a family is allocated an array with index range "1..1". Element -1 of ENTRY_ARRAY is dedicated to an entry that is associated with all TERMINATE alternatives appearing in SELECT statements; element 0 is used by DELAY statements.

3.2.3.1.2 Stack Region

Each task has a stack region within which its data are stored. The stack is a contiguous block of memory that grows from the low end of logical address space toward the high end. The stack region is subdivided into stack frames, each of which contains the local storage associated with the activation of a subprogram. This storage includes linkage information, parameters, variables of static size, and variables of dynamic size.

3.2.3.1.3 Task Scheduling Block

This control block is used to implement the Ada task scheduling policy. It functionally contains a list of task control blocks ordered by descending priority; the head of the list is the most urgent task, the one that is executing. The actual implementation has a pointer to the active task and a pointer to the ordered list of tasks that are ready to execute but do not have great enough urgency to preempt the active task; this representation can be extended to tightly coupled multi-processor configurations.

3.2.3.1.4 Entry Descriptor

One task requests that another provide a service by calling an entry of the server. Within the server, a willingness to respond to a particular request is indicated by executing an ACCEPT statement for the associated entry. The server may wait on any one of several requests by executing a SELECT statement that contains several ACCEPT statements. The ACCEPT statement for an entry is executed in a state called rendezvous that ensures exclusive access to the resources of the task.

Each entry is represented by a descriptor that is constructed during task elaboration and contains the following fields.

1. OPEN_ENTRY -- This field is not NULL if and only if this entry is "open"; that is, the entry corresponding to this descriptor has been the operand of an ACCEPT statement. If this field is not NULL, it contains a link in the circularly linked list of all entry descriptors that are simultaneously open in this task.

2. WAITING_TASK_LIST -- This field points to the tail of a circularly-linked, first-in, first-out queue of task control blocks for tasks that have called this entry but have not been accepted for rendezvous.

3. CODE_ADDRESS -- This field contains the address of the instruction at which execution begins for the entry associated with this descriptor. Since more than one ACCEPT statement may be used for a given entry, this address cannot be determined until an ACCEPT is executed.

3.2.3.1.5 An Example Rendezvous

The following five figures illustrate how these data structures are manipulated during a rendezvous.

Figure 3-15 shows task A is active and tasks B, C, and D are ready to execute. Task A has entry descriptors for its TERMINATE alternative and entries X and Y; task E has called entry Y but is waiting on its entry descriptor since the entry is not open. If task A executes the statement

```
select
  accept X( I ) do
    -- Statement_1
  end;
  -- Statement_2
or
  terminate
end select;
```

the system goes to the state reflected in Figure 3-16. Task A has entered entry X and the TERMINATE alternative in its open entry list and suspended itself by a semaphore P operation on its SUSPEND semaphore. Task B has moved from the head of the ready queue to become the active task. Figure 3-17 shows what happens if task B calls entry X of task A. Since that entry is open, rendezvous may occur immediately; code that executes within B performs the following functions:

1. Set field ARG_LIST of the task control block of B to point to argument I of the entry call.
2. Insert the task control block of B into the rendezvous partner list of A.
3. Empty the open entry list of task A.
4. Store a pointer to the entry descriptor of X in field SELECTED_ENTRY of the task control block of A.

When task A is activated from B, it becomes the active task and branches to the address that is specified in the code address field of its selected entry. Figure 3-18 shows the state of the system as task A executes Statement_1 within the ACCEPT body for X. At exit from the ACCEPT body, the rendezvous terminates, and task B is rescheduled (Figure 3-19).

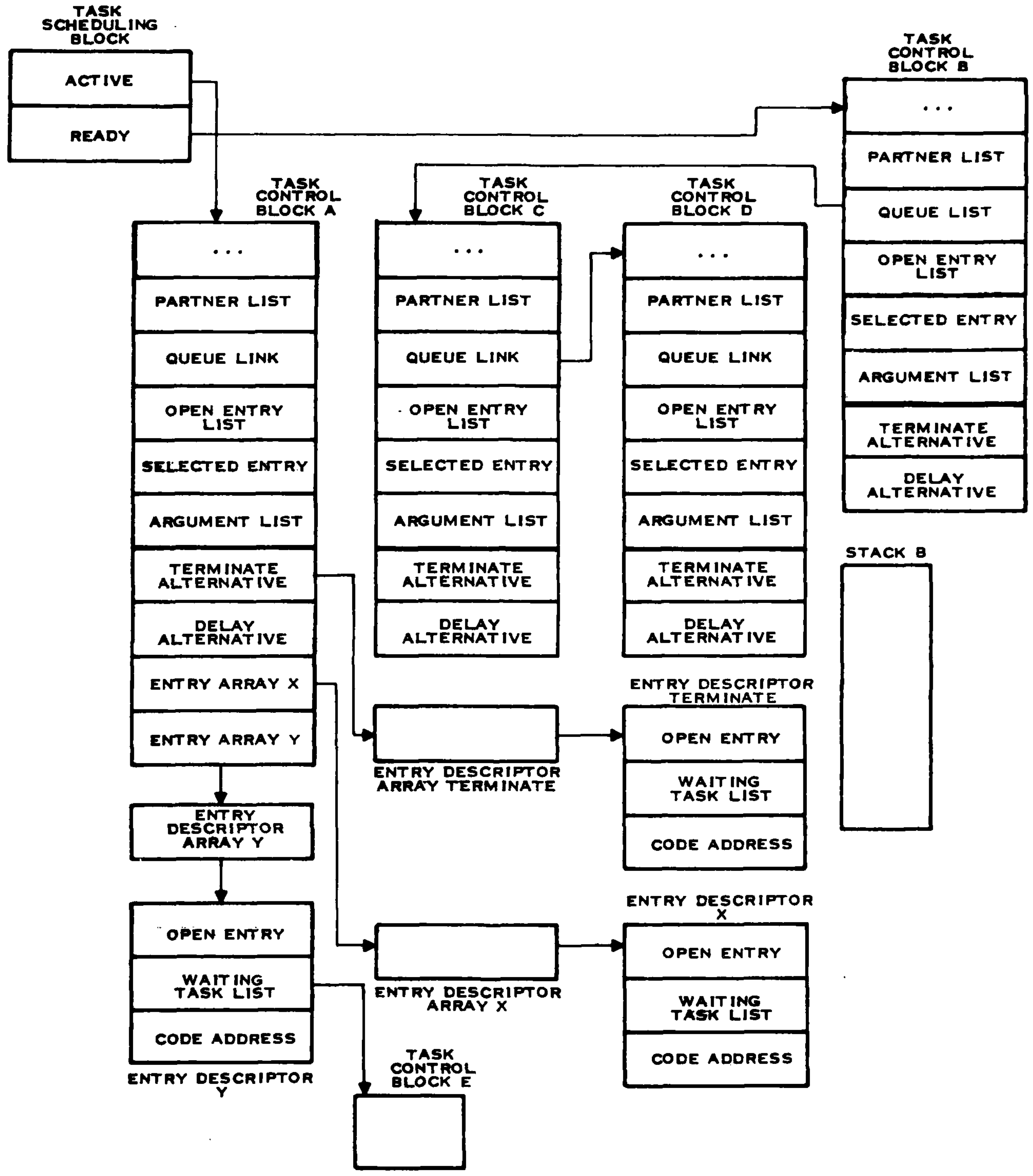


Figure 3-15 Task A Active

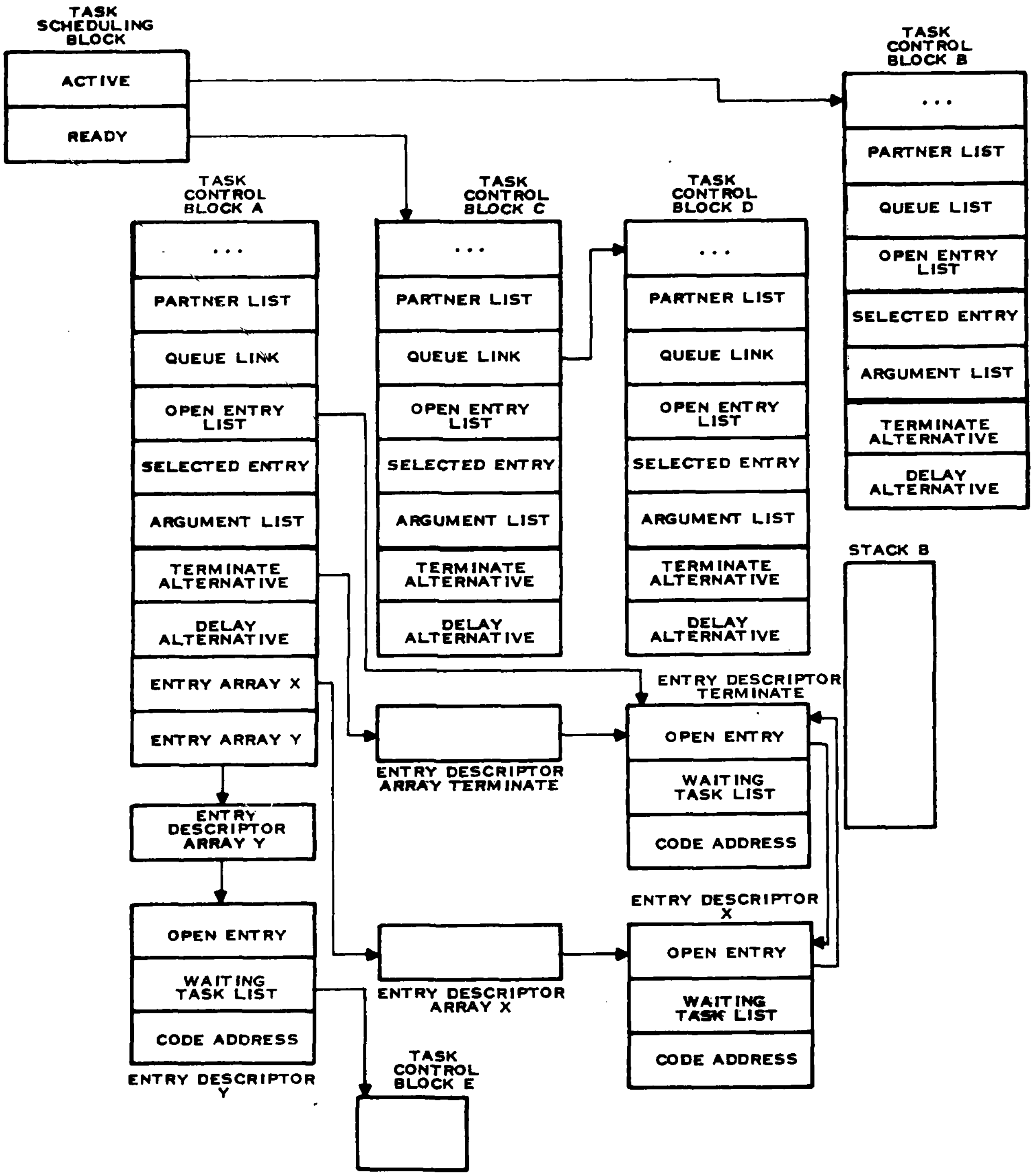


Figure 3-16 Task A Executes a SELECT Statement

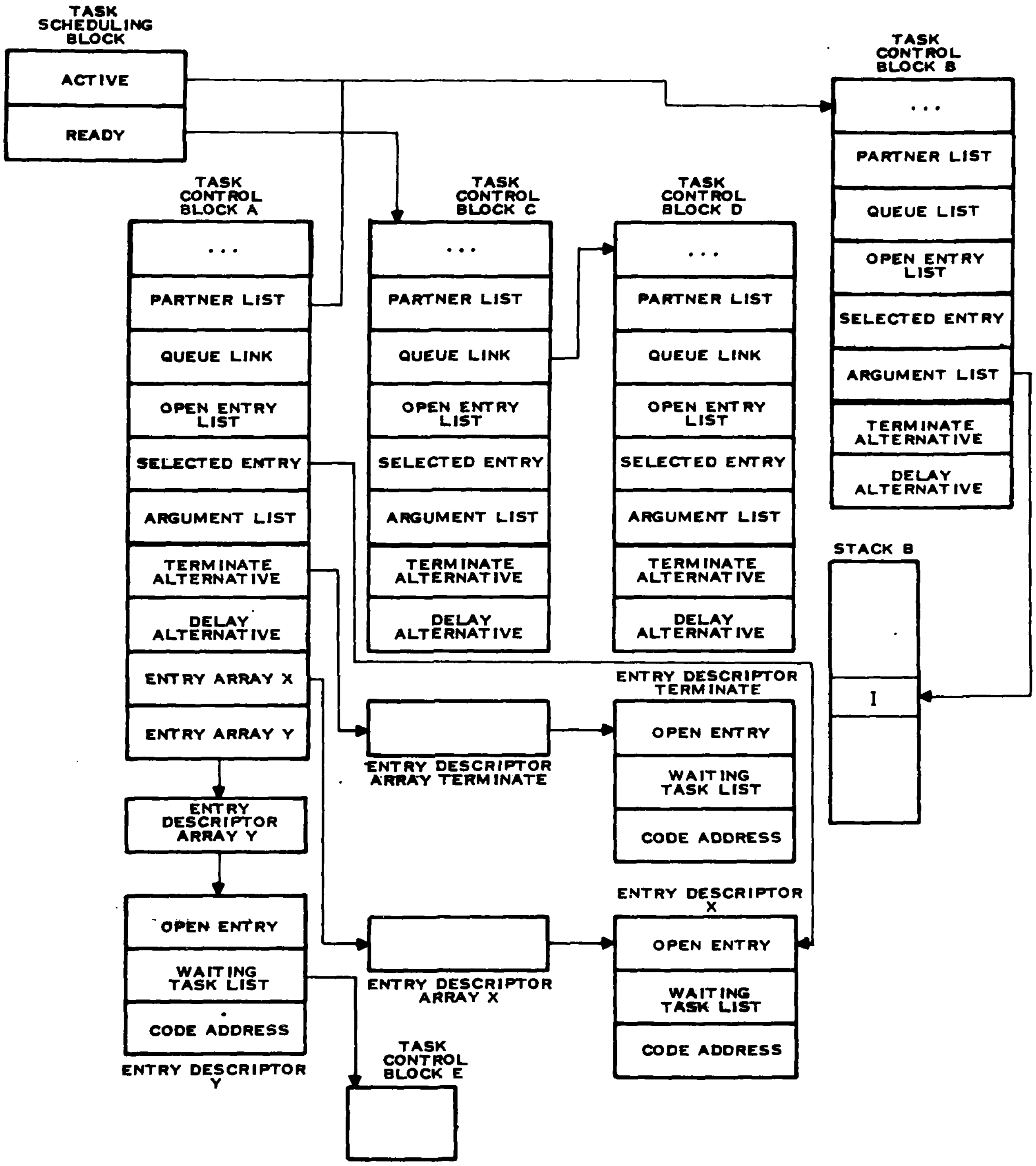


Figure 3-17 Task B Calls Entry A.X

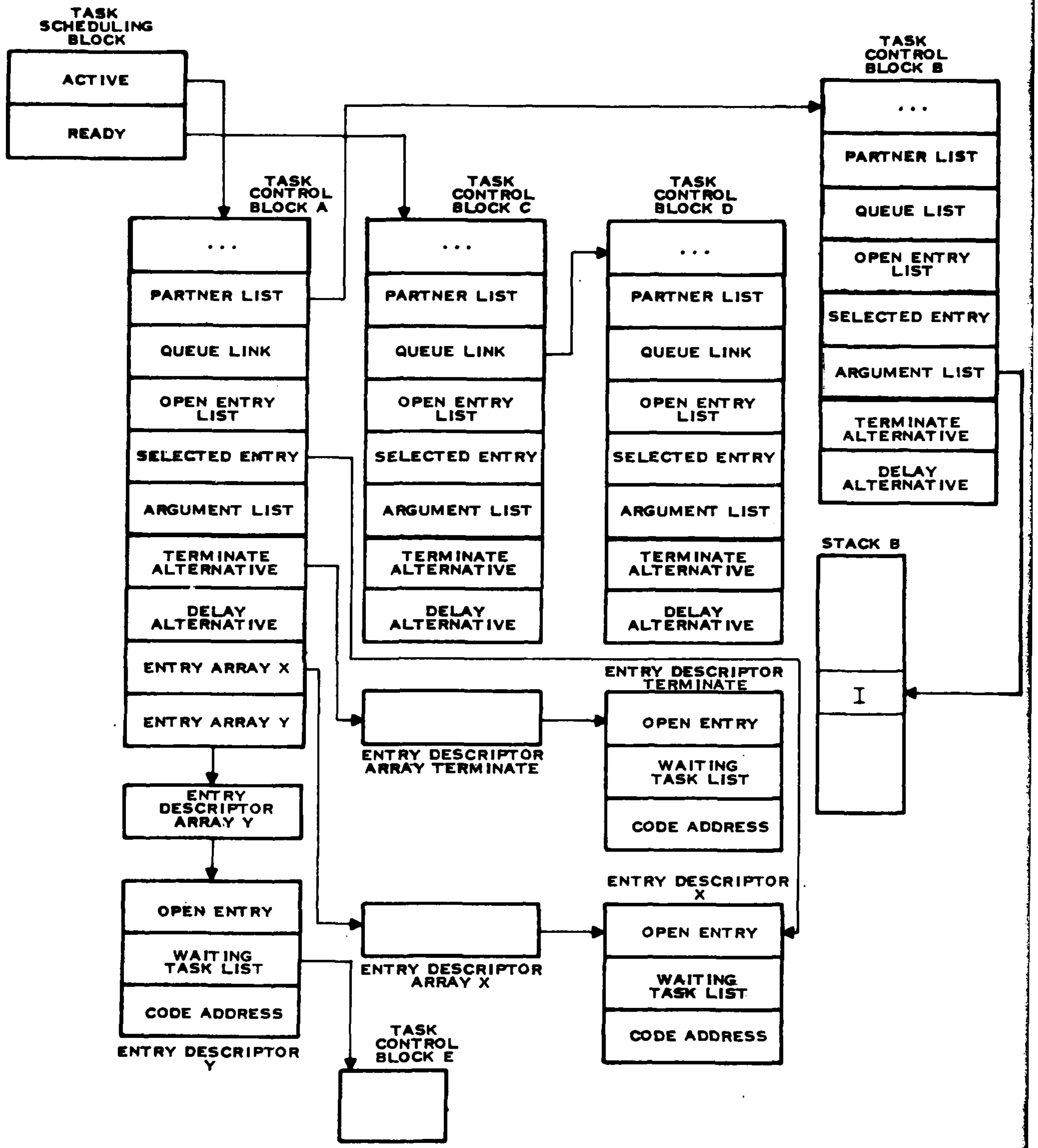


Figure 3-18 Task A Executes ACCEPT Body for Entry X

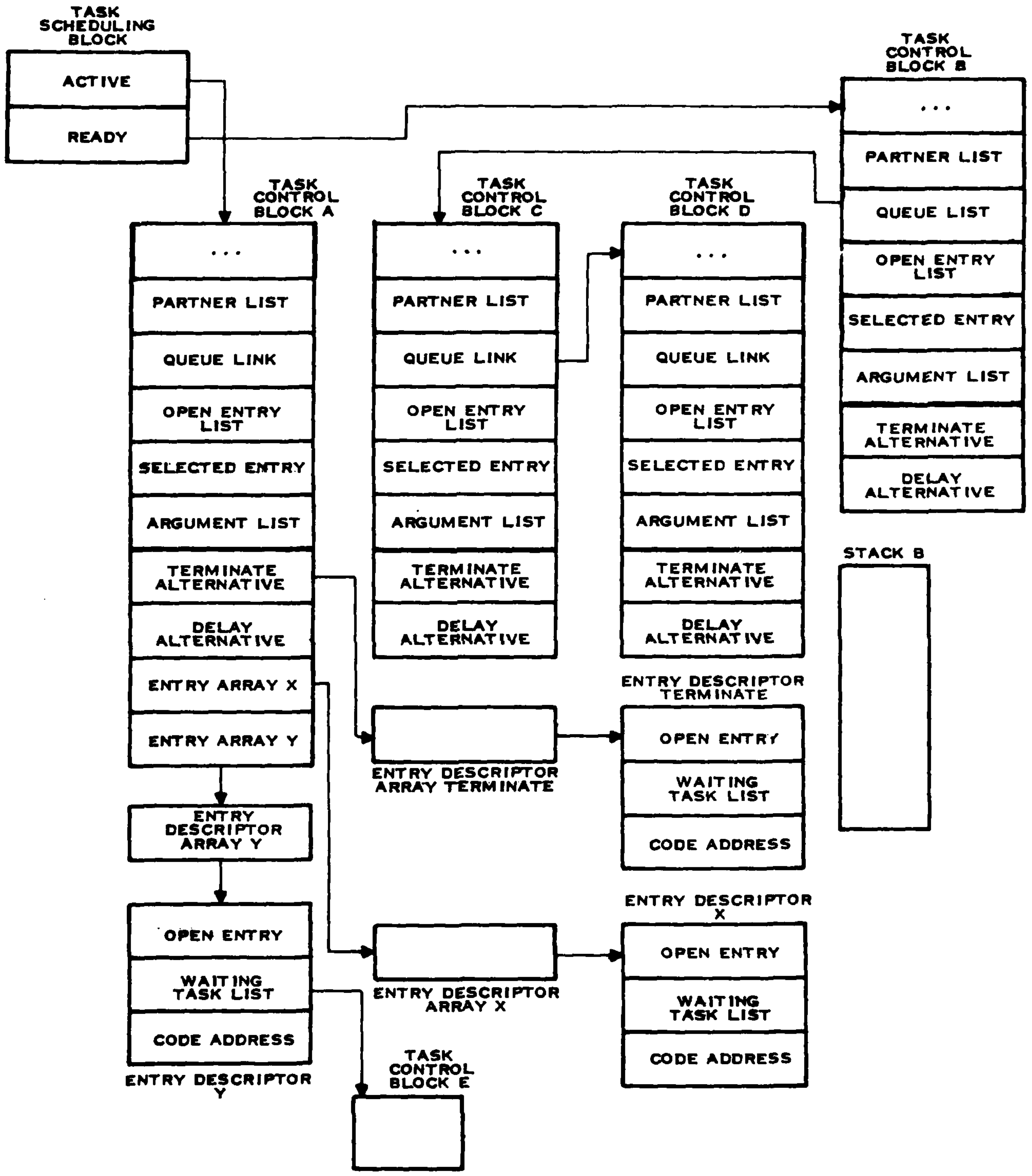


Figure 3-19 Task A Exits ACCEPT Body

3.2.3.2 Processing

3.2.3.2.1 Task Activation

The activation of a task occurs when the declarative part of the corresponding task body is elaborated; after this elaboration has been completed, the task becomes a parallel site of execution. The following steps are used to activate a task:

1. The task body associated with a task type is translated as a function subprogram that contains object code to:

   a. call task management routines to construct the data structures associated with the task;

   b. return the address of the task control block as the value associated with the task object;

   c. elaborate the declarative part of the task body;

   d. execute the statements of the task body.

2. Elaboration of a declarative part in which the task object is declared (or execution of the allocator NEW for an access type associated with the task type) causes a call to this function subprogram, which begins execution within the stack region of the task in which the declarative part occurs. (Note: the task performing the activation need not be aware of the characteristics of new tasks since they are hidden within the function subprogram.)

3. Execution in the function subprogram begins with calls to task management routines that allocate and initialize the task control block, stack region, and entry descriptors for the task.

4. A special routine is called that temporarily switches execution to the stack region of the new task.

5. Execution in the function subprogram continues (in the new stack region) with code to elaborate the declarative part of the task body. (Nonstandard elaboration code is not required since execution is in the stack region in which the object are to reside.) If an exception occurs during the step, the new task and any tasks subsequently activated within this unit are placed in a terminated state.

6. Execution in the function subprogram concludes with a call to a special routine that completes the activation of the new task and schedules it for execution:

   a. switch the stack region back to the task in which activation is being performed;

b. store the address of the return point in the function subprogram as the saved program counter value in the task control block of the new task, thus causing the task to begin execution at that point;

c. return the address of the new task control block as the value of the activated task object;

d. schedule the new task for execution;

e. adjust the return point of this routine to be the exit point of the function subprogram, thus skipping execution of the remainder of the task body;

f. return.

7. Store the result of the function as the value of the task object.

8. Call a task management routine to link the task control block of the new task into the list of all dependent tasks of the particular unit upon which the task depends; this list is maintained through a field in the linkage adminstration area associated with each unit. (Note: this operation requires exclusive access to the dependent task list.)

3.2.3.2.2 Exit From a Unit

Following [DoD80B] and [EVA80], let U be a unit that is either a block, subprogram body, task body, or library package. Define a task to be dependent on U if the task is (a) declared in U (including within an inner package but excluding within any inner block, subprogram body, or task body) or (b) designated by a value of an access type that is declared in U (including within an inner package but excluding within any inner block, subprogram body, or package body). Define the S-set of a unit U to be all the tasks dependent on U, or dependent on the body of one of those tasks, etc. A task is said to be terminated if control reaches its END statement and all of the tasks in its S-set are also terminated. Finally, a task is said to be potentially terminated if it is (a) terminated, (b) waiting at its END statement (for tasks in its S-set to terminate), or (c) waiting at a SELECT statement for which there is an open TERMINATE alternative. (Note: the S-set of a unit does not contain tasks that are dependent on units of dependent tasks if these units are not task bodies. This is sufficient since S-sets will be used to determine if tasks are potentially terminated, a property that can hold only if a task is executing within its body.)

A unit U may not be exited until all tasks in its S-set are potentially terminated. This rule is enforced using information that is maintained by the task management package in the linkage area of each unit's stack frame. The routine that exits from a unit performs the following steps:

1. If the dependent task list is empty (i.e., its pointer is NULL),

exit the unit. (If the field is NULL, all tasks that were dependent upon this unit have terminated, and the field cannot be changed in the future. The field can be examined for NULL without locking out dependent tasks if the list insertion and deletion algorithms are coded to never place an intermediate NULL value in the list pointer. It follows that the overhead associated with this step, which corresponds to the most likely case, is insignificant.)

2. If the pointer to the dependent task list is not NULL, get exclusive access to the list by performing a P operation on the mutual exclusion semaphore MUTEX that is allocated in the task control block and governs access to data structures of this task.

3. Check the list pointer again. If it is now NULL, perform a V operation on the mutual exclusion semaphore MUTEX and exit the unit. (The list became empty between the time it was checked in step 1 and the time exclusive access was obtained.)

4. The dependent task list is not empty; examine the S-set of the unit to determine if each of its members is potentially terminated. (This is naturally a recursive list traversal process that will be converted to an iterative one by augmenting the task control block with fields into which the state of the traversal can be stored.)

5. If some member of the S-set is not potentially terminated, proceed to the next step. Otherwise, force each member that is waiting on an open TERMINATE alternative to select that alternative, and proceed to the next step. (If a task is in an S-set for which each member is potentially terminated and if that task does not have dependents, the task must be waiting on an open TERMINATE alternative. It follows that selecting each open TERMINATE alternative will force each task in the S-set to terminate.)

6. It is not possible to terminate at this time -- either some task in the S-set of this unit is not potentially terminated or this unit must wait for the effects of the termination of some dependent task to propagate to through the S-set. The task in which this unit is executing must relinquish the processor by (a) marking its state as waiting at an END statement, (b) performing a V operation to release access to the dependent task list, and (c) performing a P operation on the semaphore SUSPEND in the task control block that is used for task suspension. When this task is reactivated (by a member of its S-set that becomes potentially terminated), return to step 1.

Implementation of this algorithm involves non-trivial synchronization issues. For example, determining that each member of an S-set is potentially terminated requires that each task in the set be suspended (in some sense) so it does not change state after being examined. In particular, a task that is waiting on an open TERMINATE alternative must not be allowed to accept another SELECT alternative while the S-set is being examined. One solution is to recognize that most examinations of an S-set will find that it is not potentially terminated and to optimize the algorithm accordingly. This can be done by making two passes over the set. The first is made without lockout to determine if some task is not potentially terminated. If such a task is not found, a second pass must be made with all tasks in a steady state to verify that they are simultaneously potentially terminated. (If the iterative algorithm is used to traverse the S-set, a semaphore must be used to get exclusive access to the auxiliary traversal fields in the task control blocks.) Care must also be taken to ensure that the termination of any dependent task results in its parent being reactivated to examine its S-set; otherwise, it is possible to cause a deadlock: a unit could find exactly one dependent task that is not terminated when the first pass is made through the S-set, but the task could terminate before the unit could suspend itself to await termination.

3.2.3.2.3 Task Termination

The resources associated with a task must be reclaimed when the task becomes terminated, i.e., execution reaches the end of its task body and all of its dependent (S-set) tasks have also terminated. (If the task body selects the TERMINATE alternative of a SELECT statement, all dependent tasks will be terminated.) This reclamation is implemented by having the task perform a normal block exit (to ensure that all dependent tasks have terminated) to a task management routine that was established as the "return" point of the task body when the task data structures were created.

There are two choices for the implementation of the task termination routine. In an environment in which there is a separate task available to provide ASE services, the routine can send a message to the ASE task asking that the structures of the terminating task be reclaimed. If such a task is not available and deallocation of a heap structure is simple (e.g., reset the ALLOCATED bit in a packet descriptor), it is possible to use the technique implemented in Microprocessor Pascal [BAT79, TI81C]:

> mask all interrupts,
>
> execute the heap deallocation algorithm using data that are held entirely in registers (i.e., not in the stack region which is one of the structures that must be deallocated),
>
> and
>
> transfer control to the task with highest priority.

3.2.3.2.4 Unconditional Entry Call

A entry E of task T is called by a statement of the form

```
T.E( ARG_1, ..., ARG_n )

or

T.E( INDEX )( ARG_1, ..., ARG_n )
```

where the second form is used for a family of entries. Such calls are translated by the compiler into one of

```
ENTRY_CALL( ARG_LIST, TCB, ENTRY_NUMBER )

ENTRY_CALL( ARG_LIST, TCB, ENTRY_NUMBER, INDEX )
```

where ARG_LIST is the address of an argument list containing ARG_1, ..., ARG_n, TCB is the address of the task control block of T, and ENTRY_NUMBER is the index of E in the entry array of T. The two versions of ENTRY_CALL are machine language routines that perform a vectored call to a task entry. (If the entry is a family, its entry descriptor array contains more than on entry descriptor address and a specific member is selected by INDEX.) Figure 3-20 shows the algorithm for the version of ENTRY_CALL that is used for an entry that is a member of a family.

```
procedure ENTRY_CALL(
          ARG_LIST     : ADDRESS_RANGE;
          TCB          : TASK_CONTROL_BLOCK_ADDRESS;
          ENTRY_NUMBER : ENTRY_RANGE;
          INDEX        : INTEGER ) is
  ED : ENTRY_DESCRIPTOR_ADDRESS;
begin
  P( TCB.MUTEX );  -- Get exclusive access to task structures.
  MY_TCB().ARG_LIST := ARG_LIST;
  ED := TCB.ENTRY_ARRAY( ENTRY_NUMBER )( INDEX );
  if ED.OPEN_ENTRY /= null
    then -- ACCEPT preceded ENTRY call.
    -- Delete all entries from TCB.OPEN_ENTRY_LIST.
    -- Enqueue MY_TCB() at the head of TCB.PARTNER_LIST.
    MY_TCB().SAVED_PRIORITY := TCB.PRIORITY;
    -- Set rendezvous priority in TCB.PRIORITY.
    TCB.SELECTED_ENTRY := ED;
    V( TCB.SUSPEND ); -- Activate T.
    P( MY_TCB().SUSPEND ); -- Wait for rendezvous to complete.
  else -- ENTRY call preceded ACCEPT.
    -- Insert MY_TCB() at tail of ED.WAITING_TASK_LIST.
    V( TCB.MUTEX ); -- Release exclusive access.
    P( MY_TCB().SUSPEND ); -- Wait for rendezvous to complete.
  end if;
end ENTRY_CALL;
```

Figure 3-20 Procedure ENTRY_CALL

The address of the argument list of the task entry is saved in ARG LIST field of the task control block MY_TCB() associated with the calling task; the function ENTRY_ARGUMENT_LIST will be used by the compiler to retrieve this address and access arguments. (MY_TCB() is an inline function that returns the address of the task control block of the active task.) There are two cases to be considered based on the relation of this entry call to the execution of an ACCEPT statement for entry E of T.

If the field OPEN_ENTRY of the entry descriptor ED of E is NULL, the calling task must be suspended to await the execution in T of an ACCEPT statement for E. The task is placed in the queue for the entry E by inserting its task control block at the tail of ED.WAITING_TASK_LIST. A V operation is performed on the mutual exclusion semaphore MUTEX of T to relinquish exclusive access to the task management data structures of T. A P operation is performed on the SUSPEND semaphore of the calling task to relinquish the processor; when control returns, the rendezvous will be complete.

If the field OPEN_ENTRY of the entry descriptor ED of E is not NULL, an ACCEPT statement for E has been executed in T, and the rendezvous of the calling task and T may begin. Subsequent entry calls are disabled by emptying the list of all open entries of T, which has the effect of setting to NULL both the OPEN_ENTRY_LIST field of T's control block and the OPEN_ENTRY field of each entry descriptor of T. The task control block of the calling task is inserted at the head of the rendezvous PARTNER_LIST of T, the last-in, first-out queue which is used by the function ENTRY_ARGUMENT_LIST to access entry arguments and by T to determine whom to reactivate at the conclusion of rendezvous. The SAVED_PRIORITY field of the calling task's control block is used to save the priority of T so it can be adjusted for the duration of the rendezvous according to the rules given in [DoD80B, 9.8]. In case T has executed a SELECT statement, the address of the entry descriptor of E is stored in SELECTED_ENTRY so the procedure SELECT_WAIT can return to the appropriate ACCEPT body. Task T is activated via a V operation on the semaphore SUSPEND upon which it is suspended, and the calling task performs a P operation on its SUSPEND semaphore to await conclusion of the rendezvous. Observe that exclusive access to the data structures of T is passed to entry E [HAB80]; this ensures that no other task waiting on TCB.MUTEX can intervene. Access is relinquished in the procedure in which T is waiting, which is BEGIN_ACCEPT or SELECT_WAIT.

Both cases described above conclude with a V operation that activates another task and a P operation to suspend the calling task. Both cases are suitable for optimization. In the first case, it is possible that relinquishing access to the structures of T will activate a task with a high enough priority to preempt the calling task before it can execute the P operation to suspend. This results in unnecessary task scheduling activity that can be prevented by providing a semaphore operation V_P that performs an indivisible V and P operation pair. This new operation can also be used in the second case. However, that case can be optimized further since the Ada rules for task priority during rendezvous permit the implementation of a scheduling policy for which the activation of T and suspension of the calling task can be made as an exchange of the processor, without regard to other tasks that are waiting to execute. Thus, it is possible to optimize the last step to not require a full context switch with invocation of the scheduler.

3.2.3.2.5 Conditional Entry Call

This form of entry call is translated into a call to a function similar to ENTRY_CALL that does not wait if a rendezvous is not immediately possible and returns a Boolean result that indicates if the rendezvous was made. The statement

```
select
  T.E( INDEX )( ARG_1, ..., ARG_n );
  -- Statements if immediate rendezvous.
else
  -- Statements if no rendezvous.
end select;
```

is translated as

```
if CONDITIONAL_ENTRY_CALL( ARG_LIST,
     ENTRY_NUMBER, INDEX, MAX_TIME ) then
  -- Statements if immediate rendezvous.
else
  -- Statements if no rendezvous.
end if;
```

The algorithm for CONDITIONAL_ENTRY_CALL is given in Figure 3-21.

```
function CONDITIONAL_ENTRY_CALL(
           ARG_LIST      : ADDRESS_RANGE;
           TCB           : TASK_CONTROL_BLOCK_ADDRESS;
           ENTRY_NUMBER  : ENTRY_RANGE;
           INDEX         : INTEGER ) is
    return BOOLEAN is
  ED : ENTRY_DESCRIPTOR_ADDRESS;
begin
  P( TCB.MUTEX );  -- Get exclusive access to task structures.
  MY_TCB().ARG_LIST := ARG_LIST;
  ED := TCB.ENTRY_ARRAY( ENTRY_NUMBER )( INDEX );
  if ED.OPEN_ENTRY /= null
    then -- ACCEPT preceded ENTRY call; rendezvous is possible.
    -- Delete all entries from TCB.OPEN_ENTRY_LIST.
    -- Enqueue MY_TCB() at the head of TCB.PARTNER_LIST.
    MY_TCB().SAVED_PRIORITY := TCB.PRIORITY;
    -- Set rendezvous priority in TCB.PRIORITY.
    TCB.SELECTED_ENTRY := ED;
    V( TCB.SUSPEND ); -- Activate T.
    P( MY_TCB().SUSPEND ); -- Wait for rendezvous to complete.
    return TRUE;
  else -- ENTRY call preceded ACCEPT; no rendezvous.
    V( TCB.MUTEX );
    return FALSE;
  end if;
end CONDITIONAL_ENTRY_CALL;
```

Figure 3-21 Function CONDITIONAL_ENTRY_CALL

3.2.3.2.6 Timed Entry Call

This form of entry call is translated into a call to a function TIMED_ENTRY_CALL that waits at most the specified time for a rendezvous and returns a Boolean result that indicates if the rendezvous was made. The statement

```
select
  T.E( INDEX )( ARG_1, ..., ARG_n );
  -- Statements if rendezvous.
else
  delay MAX_TIME;
  -- Statements if time-out before rendezvous.
end select;
```

is translated as

```
if TIMED_ENTRY_CALL( ARG_LIST, TCB
                     ENTRY_NUMBER, INDEX,
                     MAX_TIME ) then
  -- Statements if rendezvous.
else
  -- Statements if time-out before rendezvous.
end if;
```

The algorithm for TIMED_ENTRY_CALL, which is shown in Figure 3-22, is similar to that for CONDITIONAL_ENTRY_CALL except for the provision for a time-out. If rendezvous cannot begin immediately and MAX_TIME is in the future, execution must be suspended until either rendezvous or a time-out occurs. A request is made to the clock manager to cancel the last timed entry call if it has not yet timed out. The time at which reactivation must occur is calculated in the ENTRY_TIME field of the current task control block, and a request is made to the clock manager to schedule reactivation. The active task control block is queued on the descriptor of E, exclusion is released, and the task waits for a change of state. If time-out occurs before rendezvous, the clock manager sets ENTRY_TIME to 0.0 and dequeues the calling task's control block from the descriptor of E. If rendezvous occurs first, no further processing is required. In particular, a message is not sent to the clock manager to cancel the timed entry call; cancellation will occur the next time this task requests a timed entry call if time-out has not occurred at that time.

```
function TIMED_ENTRY_CALL(
           ARG_LIST     : ADDRESS_RANGE;
           TCB.         : TASK_CONTROL_BLOCK_ADDRESS;
           ENTRY_NUMBER : ENTRY_RANGE;
           INDEX        : INTEGER;
           MAX_TIME     : DURATION ) is
    return BOOLEAN is
  ED : ENTRY_DESCRIPTOR_ADDRESS;
begin
  P( TCB.MUTEX );  -- Get exclusive access to task structures.
  TCB.ARG_LIST := ARG_LIST;
  ED := TCB.ENTRY_ARRAY( ENTRY_NUMBER )( INDEX );
  if ED.OPEN_ENTRY /= null
    then -- ACCEPT preceded ENTRY call.
    -- Delete all entries from TCB.OPEN_ENTRY_LIST.
    -- Enqueue MY_TCB() at the head of TCB.PARTNER_LIST.
    MY_TCB().SAVED_PRIORITY := TCB.PRIORITY;
    -- Set rendezvous priority in TCB.PRIORITY.
    TCB.SELECTED_ENTRY := ED;
    V( TCB.SUSPEND ); -- Activate T.
    P( MY_TCB().SUSPEND ); -- Wait for rendezvous to complete.
    return TRUE;
  else -- ENTRY call preceded ACCEPT.
    if MAX_TIME <= 0 then
      V( TCB.MUTEX );
      return FALSE;
    else
      if MY_TCB().ENTRY_TIME /= 0.0 then
        -- Request Clock Manager to cancel timed entry call.
      end if;
      MY_TCB().ENTRY_TIME := CURRENT_TIME() + MAX_TIME;
      MY_TCB().SELECTED_ENTRY := ED;
      -- Insert MY_TCB() at tail of ED.WAITING_TASK_LIST.
      -- Request Clock Manager to schedule a timed entry
      --   reactivation an MY_TCB().ENTRY_TIME.
      V( TCB.MUTEX ); -- Release exclusive access.
      P( MY_TCB().SUSPEND ); -- Wait for reactivation.
      if MY_TCB().ENTRY_TIME = 0.0 then
        -- Assert: Clock Manager removed MY_TCB() from
        --         TCB.ED.WAITING_TASK_LIST.
        return FALSE;
      else
        return TRUE;
      end if;
    end if;
  end if;
end TIMED_ENTRY_CALL;
```

Figure 3-22 Function TIMED_ENTRY_CALL

3.2.3.2.7 Interrupt Handling

If an address specification is given for an entry of a task, that entry is associated with an interrupt whose occurrence causes a call to the entry. Implementation of interrupt handling is made through a table that maps interrupt address specifications to task entries. The compiler elaborates an entry with address specification by calling

```
procedure CONNECT_INTERRUPT(
               INTERRUPT_ADDRESS : ADDRESS_RANGE;
               ENTRY_NUMBER      : ENTRY_RANGE )
```

where ENTRY_NUMBER is the number of the entry in its task. This procedure points the table entry for the specified interrupt to the descriptor for the task entry. When an interrupt occurs, machine-specific code converts the interrupt into an entry call. This code will have to be aware of any arguments supplied by the interrupt and place them in a list where they can be accessed by the ACCEPT body associated with the entry. If the protocol for an interrupt requires some form of acknowledgment, it will have to be supplied by this interface code prior to activation of the entry.

3.2.3.2.8 ACCEPT Statement

The body of an ACCEPT statement that does not occur within a SELECT alternative is translated by preceding the statements of the body with a call procedure BEGIN_ACCEPT and following them with a call to END_ACCEPT. (The translation of ACCEPT statements as a SELECT alternative is discussed in the section on SELECT statements.)

Consider an ACCEPT statement for an entry E that has ordinal number ENTRY NUMBER among the entries of task T. There are two versions of BEGIN ACCEPT

```
procedure BEGIN_ACCEPT( ENTRY_NUMBER : ENTRY_RANGE )

procedure BEGIN_ACCEPT( ENTRY_NUMBER : ENTRY_RANGE;
                        INDEX        : INTEGER )
```

where the second is used if the E is a member of a family and has index INDEX. The algorithms for BEGIN_ACCEPT and END_ACCEPT are given in Figure 3-23 and Figure 3-24, respectively.

There are two cases to be considered in BEGIN_ACCEPT. If WAITING_TASK_LIST is empty, the ACCEPT statement has occurred before an ENTRY call. In this case, the descriptor ED for the entry is inserted into the open entry list of the current task to mark the entry as "open," which means it is receptive to entry calls. The task waits for an entry call by releasing mutual exclusion and suspending on its SUSPEND semaphore. Reactivation occurs when E is called from another task; the call V(MY_TCB().MUTEX) must be made to relinquish the exclusive access to task data structures that was passed from the rendezvous partner. (There is no need to use ED.CODE_ADDRESS as the return point since this ACCEPT did not occur in a SELECT.) If WAITING_TASK_LIST is not empty, an entry call preceded the ACCEPT statement, and rendezvous may begin immediately. The task control block of the calling task is dequeued from the head of WAITING_TASK_LIST and inserted at the head of PARTNER_LIST. The priority of the active task is adjusted for the rendezvous, mutual exclusion is released, and control returns to the ACCEPT body.

```
procedure BEGIN_ACCEPT( ENTRY_NUMBER : ENTRY_RANGE;
                        INDEX        : INTEGER ) is
  ED  : ENTRY_DESCRIPTOR_ADDRESS;
  TCB : TASK_CONTROL_BLOCK_ADDRESS;
begin
  P( MY_TCB().MUTEX );
  ED := MY_TCB().ENTRY_ARRAY( ENTRY_NUMBER )( INDEX );
  if "ED.WAITING_TASK_LIST is empty"
    then -- ACCEPT preceded ENTRY call.
    -- Insert ED into MY_TCB().OPEN_ENTRY_LIST.
    V( MY_TCB().MUTEX );
    P( MY_TCB().SUSPEND );
    V( MY_TCB().MUTEX );
  else -- ENTRY call preceded ACCEPT.
    -- Dequeue TCB from head of ED.WAITING_TASK_LIST.
    -- Enqueue TCB at head of MY_TCB().PARTNER_LIST.
    TCB.SAVED_PRIORITY := MY_TCB().PRIORITY;
    -- Set rendezvous priority in MY_TCB().PRIORITY.
    V( MY_TCB().MUTEX );
  end if;
end BEGIN_ACCEPT;
```

Figure 3-23 Procedure BEGIN_ACCEPT

The processing in END_ACCEPT is straightforward: remove the task control block of the rendezvous partner for the head of PARTNER_LIST, restore the priority of the current task, and reactivate the partner.

```
procedure END_ACCEPT is
  TCB : TASK_CONTROL_BLOCK_ADDRESS;
begin
  -- Dequeue head of MY_TCB().PARTNER_LIST into TCB.
  MY_TCB().PRIORITY := TCB.SAVED_PRIORITY;
  V( TCB.SUSPEND );
end END_ACCEPT;
```

Figure 3-24 Procedure END_ACCEPT

Within the body of an ACCEPT statement, arguments of the associated entry must be referenced indirectly using a pointer that has been returned by function ENTRY_ARGUMENT_LIST (Figure 3-25); the argument NESTING_LEVEL permits access to arguments of nested ACCEPT statements. (The compiler should move an argument list pointer into a local variable if it will be used more than once. The argument list pointer for the inner most ACCEPT could also be returned in the standard parameter list register.)

Observe that mutual exclusion is not in effect while the body of an ACCEPT statement is executing. Exclusion is not needed since the only way another task can enter rendezvous with this task is for some entry to be placed in the OPEN_ENTRY_LIST, which can occur only if another ACCEPT or SELECT statement is executed from within the current task. (In fact, execution with exclusive access would lead to deadlock if a nested ACCEPT or SELECT occurred.)

```
function ENTRY_ARGUMENT_LIST(
         NESTING_LEVEL : NONNEGATIVE )
  return ADDRESS_RANGE is
begin
  -- Return the ARG_LIST field of the task control
  --   block that is NESTING_LEVEL blocks behind
  --   the head of the list MY_TCB().PARTNER_LIST.
end ENTRY_ARGUMENT_LIST;
```

Figure 3-25 Function ENTRY_ARGUMENT_LIST

3.2.3.2.9 SELECT Statement

There are two basic translation templates for the SELECT statement; the choice between the two depends on whether an ELSE clause is present. Figure 3-26 shows that a SELECT statement without an ELSE clause is translated as a call to procedure INITIALIZE_SELECT followed by a series of nested IF statements that examine each alternative of the SELECT statement and fall through to a call to SELECT_WAIT if no alternative can be immediately selected. (The template in Figure 3-26 illustrates the translation of both DELAY and TERMINATE alternatives although both may not occur in the same SELECT statement.) When an ELSE clause is present, the translation is similar (Figure 3-27) except control falls through to a call to SELECT_ELSE and user-specified statements are executed if no alternative can be selected. The predicate of each IF statement contains the WHEN condition that guards the associated alternative and a call to a Boolean function that examines the alternative. If the function is DELAY_ALTERNATIVE or TERMINATE_ALTERNATIVE, it always returns FALSE to force the next clause of the IF statement to be examined. If the function is ACCEPT_ALTERNATIVE, two versions are supplied. If it is known for a particular application that no WHEN conditions have side effects, a version may be used that returns TRUE if an entry call is waiting for the alternative and proceeds immediately to rendezvous, thus by-passing the unnecessary evaluation of other alternatives. If this optimization is not desired by the user, a version that always returns FALSE must be used; the selection of an alternative is postponed until SELECT_WAIT or SELECT_ELSE, at which point all guards have been evaluated. (It would also be possible to have the compiler choose the version of ACCEPT_ALTERNATIVE based on the nature of the WHEN conditions of a particular SELECT statement.) If a WHEN condition is TRUE but the alternative is not immediately available, the SELECT data structures are modified to indicate that the alternative is "open" and to note the address at which the alternative must begin execution if it is selected in the future. (This address is specified by argument ADDRESS_INCREMENT, the relative displacement of the desired code from the point to which the alternative evaluation function returns.)

```
select
  when GUARD_1 =>
    accept ENTRY_NAME( INDEX )( ARG_1, ..., ARG_n ) do
      -- ACCEPT body.
    end;
    -- Statements_1.
or
  when GUARD_2 =>
    delay INTERVAL;
    -- Statements_2.
or
  when GUARD_3 =>
    terminate;
end select;

INITIALIZE_SELECT;
if GUARD_1 and then
   ACCEPT_ALTERNATIVE( ADDRESS_INCREMENT_1,
                       ENTRY_NUMBER, INDEX ) then
  -- ACCEPT body.
  END_ACCEPT;
  -- Statements_1.
elsif GUARD_2 and then
      DELAY_ALTERNATIVE( ADDRESS_INCREMENT_2,
                         INTERVAL ) then
  -- Statements_2.
elsif GUARD_3 and then
      TERMINATE_ALTERNATIVE( ADDRESS_INCREMENT_3 ) then
  -- Branch to appropriate exit point.
else
  SELECT_WAIT.
end if;
```

Figure 3-26 SELECT Statement without ELSE Clause

```
select
  when GUARD_1 =>
    accept ENTRY_NAME( INDEX )( ARG_1, ..., ARG_n ) do
      -- ACCEPT body.
    end;
    -- Statements_1.
else
  -- Statements_2.
end select;

INITIALIZE_SELECT;
if GUARD_1 and then
   ACCEPT_ALTERNATIVE( ADDRESS_INCREMENT_1,
                       ENTRY_NUMBER, INDEX ) then
  -- ACCEPT body.
  END_ACCEPT;
  -- Statements_1.
else
  SELECT_ELSE;
  -- Statements_2.
end if;
```

Figure 3-27 SELECT Statement with ELSE Clause

3.2.3.2.9.1 Procedure INITIALIZE_SELECT

This procedure, whose processing is described in Figure 3-28, must be executed at the beginning of a SELECT statement to initialize the associated data structures. A P operation is performed on the mutual exclusion semaphore MUTEX of the active task control block to lockout other tasks attempting to rendezvous. (This lockout is required to prevent an alternative that is placed in OPEN_ENTRY_LIST from being selected before the end of the SELECT statement is reached and the active task can be placed in a state in which it can respond to rendezvous requests.) If the field DELAY_TIME is not 0.0, the last SELECT statement that executed in this task had a DELAY alternative that has not timed out; a message must be sent to the clock manager to cancel the pending time-out. (The OPEN_ENTRY_LIST of the active task must be empty when evaluation of the SELECT alternatives begins because entry descriptors will be placed into this list if the corresponding alternatives are open. However, the list need not be emptied in INITIALIZE_SELECT since all entries are removed whenever rendezvous occurs.)

```
procedure INITIALIZE_SELECT is
begin
  P( MY_TCB().MUTEX );
  if MY_TCB().DELAY_TIME /= 0 then
    -- Request Clock Manager to cancel activation for
    --   DELAY alternative.
    MY_TCB().DELAY_TIME := 0;
  end if;
  -- Assert:  OPEN_ENTRY = null for each entry of this task
  --          and MY_TCB().OPEN_ENTRY_LIST = null.
end INITIALIZE_SELECT;
```

Figure 3-28 Procedure INITIALIZE_SELECT

3.2.3.2.9.2 Function ACCEPT_ALTERNATIVE

Figure 3-29 shows the version of ACCEPT_ALTERNATIVE that is used if side effects are permitted in WHEN conditions. (An overloaded variant of this function will be supplied for use with an entry that is not a member of a family.) In this case, immediate selection of an entry for which a call is waiting cannot be made, and the bulk of the processing must be deferred to SELECT_WAIT or SELECT_ELSE. If the entry descriptor ED associated with the entry E is not already in the queue OPEN_ENTRY_LIST, it is inserted to indicate that E is open, and the address of the ACCEPT body is saved in ED.CODE_ADDRESS. (The descriptor will already be in the queue if the entry was specified on a previous open alternative; this algorithm selects the first open alternative.)

```
function ACCEPT_ALTERNATIVE(
           ADDRESS_INCREMENT : ADDRESS_RANGE;
           ENTRY_NUMBER      : ENTRY_RANGE;
           INDEX : INTEGER )
    return BOOLEAN is
  ED : ENTRY_DESCRIPTOR_ADDRESS;
begin
  ED := MY_TCB().ENTRY_ARRAY( ENTRY_NUMBER )( INDEX );
  if ED.OPEN_ENTRY = null then
    -- Insert ED into MY_TCB().OPEN_ENTRY_LIST.
    -- Save "return address + ADDRESS_INCREMENT"
    --   in ED.CODE_ADDRESS.
  end if;
  return FALSE;
end ACCEPT_ALTERNATIVE;
```

Figure 3-29 Function ACCEPT_ALTERNATIVE

Figure 3-30 shows the version of ACCEPT_ALTERNATIVE that has been optimized to permit immediate rendezvous with an entry for which a call is waiting. (An overloaded variant of this function will be supplied for use with an entry that is not a member of a famil,.) WAITING_TASK_LIST of the associated entry descriptor is examined to determine if an entry call is pending. If the list is empty, no call is pending. If the entry descriptor ED is not already in the queue OPEN_ENTRY_LIST, it is inserted to indicate that E is open, and the address of the ACCEPT body is saved in ED.CODE_ADDRESS. (The descriptor will already be in the queue if the entry was specified on a previous open alternative; this algorithm selects the first open alternative for a given entry.) FALSE is returned so the next alternative will be examined. If WAITING_TASK_LIST is not empty, rendezvous can occur immediately. Any open alternatives are removed from OPEN_ENTRY_LIST so they will not cause subsequent rendezvous. The task control block at the head of WAITING_TASK_LIST is removed and inserted at the head of the last-in, first-out queue of rendezvous partners. The priority of the active task is adjusted to reflect the priority of the rendezvous, and mutual exclusion is released. TRUE is returned as the result of the function, which causes the ACCEPT body to be executed. Within the ACCEPT body, the function ENTRY_ARGUMENT_LIST is used to retrieve a pointer to the address list of an entry. The same version of END_ACCEPT is used to terminate rendezvous as was used for an ACCEPT statement not occurring within a SELECT alternative.

```
function ACCEPT_ALTERNATIVE(
           ADDRESS_INCREMENT : ADDRESS_RANGE;
           ENTRY_NUMBER : ENTRY_RANGE;
           INDEX : INTEGER )
    return BOOLEAN is
  ED  : ENTRY_DESCRIPTOR_ADDRESS;
  TCB : TASK_CONTROL_BLOCK_ADDRESS;
begin
  ED := MY_TCB().ENTRY_ARRAY( ENTRY_NUMBER )( INDEX );
  if "ED.WAITING_TASK_LIST is empty"
    then -- ACCEPT preceded ENTRY call.
    if MY_TCB().ED.OPEN_ENTRY = null then
      -- Insert ED into MY_TCB().OPEN_ENTRY_LIST.
      -- Save "return address + ADDRESS_INCREMENT"
      --   in ED.CODE_ADDRESS.
    end if;
    return FALSE;
  else -- ENTRY call preceded ACCEPT.
    -- Delete all entries from MY_TCB().OPEN_ENTRY_LIST.
    -- Dequeue TCB from head of ED.WAITING_TASK_LIST.
    -- Enqueue TCB at head of MY_TCB().PARTNER_LIST.
    TCB.SAVED_PRIORITY := MY_TCB().PRIORITY;
    -- Set rendezvous priority in MY_TCB().PRIORITY.
    V( MY_TCB.MUTEX );
    return TRUE;
  end if;
end ACCEPT_ALTERNATIVE;
```

Figure 3-30 Optimized Function ACCEPT_ALTERNATIVE

3.2.3.2.9.3 Function DELAY_ALTERNATIVE

This function, whose processing is described in Figure 3-31, determines if this is the first request for a delay (MY_TCB().DELAY_TIME = 0.0) in this SELECT statement or if the requested time of reactivation is the smallest so far. If one of these conditions is holds, the entry descriptor for the DELAY alternative is inserted into the list MY_TCB().OPEN_ENTRY_LIST (if it is not already there), and the address of the statements of the alternative is stored in ED.CODE_ADDRESS. FALSE is returned as the result of this function; the actual request for reactivation at the end of INTERVAL will be made in SELECT_WAIT, but only if no ACCEPT alternative can be selected immediately.

A DELAY alternative in a SELECT is treated as an anonymous entry (with a null ACCEPT body) so the same code can be used to revive a task following both entry calls and expired delay intervals. The second item (index 0) in the entry array is reserved for the DELAY alternative.

```
function DELAY_ALTERNATIVE(
           ADDRESS_INCREMENT : ADDRESS_RANGE;
           INTERVAL : DURATION )
    return BOOLEAN is
  ED  : ENTRY_DESCRIPTOR_ADDRESS;
  NOW : DURATION;
  TCB : TASK_CONTROL_BLOCK_ADDRESS;
begin
  NOW := CURRENT_TIME();
  if MY_TCB().DELAY_TIME = 0.0 or else
     NOW + INTERVAL < MY_TCB().DELAY_TIME
    then -- This is the smallest delay so far.
    MY_TCB().DELAY_TIME := NOW + INTERVAL;
    -- Set ED equal to the entry descriptor
    --   for the DELAY alternative:
    ED := MY_TCB().ENTRY_ARRAY( 0 )( 1 );
    if ED.OPEN_ENTRY = null then
      -- Insert ED into MY_TCB().OPEN_ENTRY_LIST.
    end if;
    -- Save "return address + ADDRESS_INCREMENT"
    --   in ED.CODE_ADDRESS.
  end if;
  return FALSE;
end DELAY_ALTERNATIVE;
```

Figure 3-31 Function DELAY_ALTERNATIVE

3.2.3.2.9.4 Function TERMINATE_ALTERNATIVE

The TERMINATE alternative of a SELECT statement is treated as an anonymous entry to which the first item (index -1) in the entry array points. This function (Figure 3-32) inserts this descriptor into the list MY_TCB().OPEN_ENTRY_LIST to indicate that a TERMINATE alternative has been opened, and the address of the termination code is stored in field CODE_ADDRESS of the descriptor.

```
function TERMINATE_ALTERNATIVE(
           ADDRESS_INCREMENT : ADDRESS_RANGE )
    return BOOLEAN is
  ED : ENTRY_DESCRIPTOR_ADDRESS;
begin
  -- Set ED to the entry descriptor for the
  --   TERMINATE alternative:
  ED := MY_TCB().ENTRY_ARRAY( -1 )( 1 );
  -- Insert entry descriptor ED for TERMINATE
  --   alternative into MY_TCB().OPEN_ENTRY_LIST.
  -- Save "return address + ADDRESS_INCREMENT"
  --   in ED.CODE_ADDRESS.
  return FALSE;
end TERMINATE_ALTERNATIVE;
```

Figure 3-32 Function TERMINATE_ALTERNATIVE

3.2.3.2.9.5 Procedure SELECT_WAIT

This procedure concludes the processing of a SELECT statement that has no ELSE clause. Two versions of this procedure are illustrated. The one in Figure 3-33 is used if side effects are permitted in WHEN conditions, in which case an alternative cannot be selected until all conditions have been evaluated. Processing begins with a test to see if some alternative is open; if not, mutual exclusion is released, and the SELECT_ERROR exception is raised. If one or more open alternatives were found that have a waiting entry call, one is selected for immediate rendezvous using the same activation sequence that was described for the optimized version of ACCEPT_ALTERNATIVE. If no entry is waiting, a test is made to determine if a DELAY alternative is open that has already expired; if that is the case, the alternative is immediately activated. If the DELAY alternative has not expired and the TERMINATE alternative is open, the parent of this task and each member of his S-set is examined; if each of these tasks is potentially terminated, each task that has an open TERMINATE alternative is forced to take it, thus causing all of the tasks to terminate. If a delay has not occurred and the TERMINATE alternative cannot be accepted, the current task must be put into a suspended state. If a DELAY alternative is open, a request is made to the clock manager for reactivation at the associated time. Mutual exclusion is released, and the task relinquishes the processor by performing a V operation on its SUSPEND semaphore. When control returns to this task, the entry specified by field MY_TCB().SELECTED_ENTRY has been selected for rendezvous (or DELAY or TERMINATE processing). Field CODE_ADDRESS of the associated entry descriptor is used to change the address to which SELECT_WAIT will return, mutual exclusion is released, and the entry is activated at procedure exit.

```
procedure SELECT_WAIT is
  ED  : ENTRY_DESCRIPTOR_ADDRESS;
  TCB : TASK_CONTROL_BLOCK_ADDRESS;
begin
  if "MY_TCB().OPEN_ENTRY_LIST is empty" then
    V( MY_TCB().MUTEX );
    raise SELECT_ERROR;
  elsif "There is some entry with descriptor ED for
           which ED.WAITING_TASK_LIST is not empty"
    then -- Select the alternative immediately.
    -- Delete all entries from MY_TCB().OPEN_ENTRY_LIST.
    -- Dequeue TCB from head of ED.WAITING_TASK_LIST.
    -- Enqueue TCB at head of MY_TCB().PARTNER_LIST.
    TCB.SAVED_PRIORITY := MY_TCB().PRIORITY;
    -- Set rendezvous priority in MY_TCB().PRIORITY.
    -- Set return address to ED.CODE_ADDRESS.
    V( MY_TCB().MUTEX );
  elsif MY_TCB().DELAY_TIME /= 0 and then
        MY_TCB().DELAY_TIME <= CURRENT_TIME()
    then -- Delay has already expired.
    -- Set ED to be the descriptor for the
    --   DELAY alternative:
    ED := MY_TCB().ENTRY_ARRAY( 0 )( 1 );
    -- Delete all entries from MY_TCB().OPEN_ENTRY_LIST.
    -- Set return address to ED.CODE_ADDRESS.
    V( MY_TCB().MUTEX );
  elsif MY_TCB().ENTRY_ARRAY( 0 )( 1 ) /= null and then
        "my parent and each member of his S-set is
         potentially terminated" then
    ED := MY_TCB().ENTRY_ARRAY( 0 )( 1 );
    -- Delete all entries from MY_TCB().OPEN_ENTRY_LIST.
    -- Set return address to ED.CODE_ADDRESS.
    -- Terminate my parent and his S-set.
    V( MY_TCB().MUTEX );
  else
    if MY_TCB().DELAY_TIME /= 0.0 then
      -- Request Clock Manager to simulate an
      --   ENTRY call at DELAY_TIME.
    end if;
    V( MY_TCB().MUTEX );
    P( MY_TCB().SUSPEND );
    -- Set return address to
    --   MY_TCB().SELECTED_ENTRY.CODE_ADDRESS.
    V( MY_TCB().MUTEX );
  end if;
end SELECT_WAIT;
```

Figure 3-33 Procedure SELECT_WAIT

The algorithm for the optimized case is the same (Figure 3-34) except there is no need to test for an open alternative with waiting entry since such an alternative would have been executed immediately by the optimized version of ACCEPT_ALTERNATIVE and control would never have reached SELECT WAIT.

```
procedure SELECT_WAIT is
  ED  : ENTRY_DESCRIPTOR_ADDRESS;
begin
  if "MY_TCB().OPEN_ENTRY_LIST is empty" then
    V( MY_TCB().MUTEX );
    raise SELECT_ERROR;
  elsif MY_TCB().DELAY_TIME /= 0.0 and then
        MY_TCB().DELAY_TIME <= CURRENT_TIME()
    then -- Delay has already expired.
    -- Set ED to be the descriptor for the
    --   DELAY alternative:
    ED := MY_TCB().ENTRY_ARRAY( 0 )( 1 );
    -- Delete all entries from MY_TCB().OPEN_ENTRY_LIST.
    -- Set return address to ED.CODE_ADDRESS.
    V( MY_TCB().MUTEX );
  elsif MY_TCB().ENTRY_ARRAY( 0 )( 1 ) /= null and then
        "my parent and each member of his S-set is
         potentially terminated" then
    ED := MY_TCB().ENTRY_ARRAY( 0 )( 1 );
    -- Delete all entries from MY_TCB().OPEN_ENTRY_LIST.
    -- Set return address to ED.CODE_ADDRESS.
    -- Terminate my parent and his S-set.
    V( MY_TCB().MUTEX );
  else
    if MY_TCB().DELAY_TIME /= 0.0 then
      -- Request Clock Manager to simulate an
      --   ENTRY call at DELAY_TIME.
    end if;
    V( MY_TCB().MUTEX );
    P( MY_TCB().SUSPEND );
    -- Set return address to
    --   MY_TCB().SELECTED_ENTRY.CODE_ADDRESS.
    V( MY_TCB().MUTEX );
  end if;
end SELECT_WAIT;
```

Figure 3-34 Optimized Procedure SELECT_WAIT

No attempt is made to cancel a DELAY alternative that has not timed-out when the task is reactivated by an entry call. To do so would require that a message be sent to the clock manager. Instead, the delay is left active with the understanding that the clock manager will set the field DELAY TIME of the task control block to 0.0 when the delay expires but will not attempt to reactivate the task if the DELAY alternative is not open. A message will be sent to the clock manager to cancel the delay only if the task enters

INTIALIZE SELECT to execute another SELECT statement and the delay is still active (DELAY_TIME is not 0.0). This approach has the potential to decrease the number of messages that must be sent to the clock manager.

3.2.3.2.9.6 Procedure SELECT_ELSE

This procedure concludes the processing of a SELECT statement having an ELSE clause. Two versions of this procedure are illustrated. The one in Figure 3-35 is used if side effects are permitted in WHEN conditions, in which case an alternative cannot be selected until all conditions have been evaluated. Processing begins with a test to see if some open alternative was found; if not, mutual exclusion is released, and a normal return is made to the statements of the ELSE clause. If one or more open alternatives were found with a waiting entry call, one is selected for immediate rendezvous; the same activation sequence is used that was described for the optimized version of ACCEPT_ALTERNATIVE. If no entry is waiting, the queue of open entries is emptied, mutual exclusion is released, and control returns to the statements of the ELSE clause.

```
procedure SELECT_ELSE is
  ED  : ENTRY_DESCRIPTOR_ADDRESS;
  TCB : TASK_CONTROL_BLOCK_ADDRESS;
begin
  if "MY_TCB().OPEN_ENTRY_LIST is not empty" then
    if "There is some entry with descriptor ED for
          which ED.WAITING_TASK_LIST is not empty"
      then -- Select the alternative immediately.
      -- Delete all entries from MY_TCB().OPEN_ENTRY_LIST.
      -- Dequeue TCB from head of ED.WAITING_TASK_LIST.
      -- Enqueue TCB at head of MY_TCB().PARTNER_LIST.
      TCB.SAVED_PRIORITY := MY_TCB().PRIORITY;
      -- Set rendezvous priority in MY_TCB().PRIORITY.
      -- Set return address to ED.CODE_ADDRESS.
    else
      -- Delete all entries from MY_TCB().OPEN_ENTRY_LIST.
      V( MY_TCB().MUTEX );
    end if;
  else
    V( MY_TCB().MUTEX );
  end if;
end SELECT_ELSE;
```

Figure 3-35 Procedure SELECT_ELSE

If side effects in WHEN conditions are not permitted and the optimized version of ACCEPT_ALTERNATIVE is used, SELECT_ELSE can be simplified as shown in Figure 3-36; in this case, an alternative with a waiting entry call would have been invoked directly by ACCEPT_ALTERNATIVE and control would not have reached SELECT_ELSE.

```
procedure SELECT_ELSE is
begin
  -- Delete all entries from MY_TCB().OPEN_ENTRY_LIST.
  V( MY_TCB().MUTEX );
end SELECT_ELSE;
```

Figure 3-36 Optimized Procedure SELECT_ELSE

3.2.3.2.10 COUNT Attribute

For an entry E of a task T, the attribute E'COUNT returns the number (entry calls that are queued for rendezvous with entry E. This attribute implemented by a call to one of

```
function ENTRY_COUNT(
            TCB          : TASK_CONTROL_BLOCK_ADDRESS;
            ENTRY_NUMBER : ENTRY_RANGER )
  return INTEGER

function ENTRY_COUNT(
            TCB          : TASK_CONTROL_BLOCK_ADDRESS;
            ENTRY_NUMBER : ENTRY_RANGER;
            INDEX : INTEGER )
  return INTEGER
```

where TCB is an access value for the task control block of T, ENTRY_NUMBEI is the ordinal number of the entry E in T, and INDEX is the index expressioi for an entry that is a member of a family. The value returned b ENTRY_COUNT is the number of task control blocks that are in the queu WAITING_TASK_LIST of the entry descriptor associated with E.

Two methods will be investigated for determining the COUNT attribute of ai entry. The preferred one is to traverse the queue of tasks waiting for ai entry call to be accepted and to count their number; however, the validit of this method depends on the details of the code that is used to insert an delete tasks from this queue. Since the COUNT attribute can be used in WHEN condition of a SELECT statement, which is evaluated within a critica section guarded by TCB.MUTEX, ENTRY_COUNT cannot use this semaphore t obtain exclusive access to the queue without causing deadlock. If the queu manipulation algorithms can be coded to never leave the queue in ai inconsistent state, this method can be used; otherwise, a field will b added to each entry descriptor in which the queue manipulation routines wil maintain the number of queue entries. (The COUNT attribute of an entry wil always be accurate if it is requested within a WHEN condition that occurs ii the task in which the entry is declared; in this case, the condition i evaluated under mutual exclusion.)

3.2.3.2.11 DELAY Statement

A DELAY statement of the form

```
delay INTERVAL;
```

requests that execution of the calling task be suspended for at least INTERVAL seconds. This processing is performed by procedure DELAY_STATEMENT (Figure 3-37), which is functionally equivalent to

```
select
  when FALSE => accept  NY_ENTRY;
or
  delay INTERVAL;
end select;
```

except a DELAY statement, unlike a SELECT statement, is not restricted to occur within a task body.

```
procedure DELAY_STATEMENT(
            INTERVAL : DURATION ) is
  ED  : ENTRY_DESCRIPTOR_ADDRESS;
begin
  if INTERVAL > 0.0 then
    P( MY_TCB().SUSPEND );
    MY_TCB().DELAY_TIME := CURRENT_TIME() + INTERVAL;
    -- Set ED equal to the entry descriptor
    --   address for the DELAY alternative:
    ED := MY_TCB().ENTRY_ARRAY( 0 )( 1 );
    -- Insert ED into MY_TCB().OPEN_ENTRY_LIST.
    -- Request Clock Manager to simulate an
    --   ENTRY call at DELAY_TIME.
    V( MY_TCB().MUTEX );
    P( MY_TCB().SUSPEND );
    V( MY_TCB().MUTEX );
end DELAY_STATEMENT;
```

Figure 3-37 Procedure DELAY_STATEMENT

3.2.3.2.12 ABORT Statement

A statement of the form

```
abort TASK_NAME_1, TASK_NAME_2;
```

causes the unconditional termination of the specified tasks and all of their dependents. The ABORT statement is translated into one call to procedure ABORT_TASK for each of the specified tasks; ABORT_TASK has one argument which is the access value for the task control block of the task to be terminated. The implementation of ABORT_TASK is to raise in the task and each of its dependents a special exception that does not match any user-defined exception or the choice OTHERS. Termination is unconditional in the sense that there is no way for the user to accept the exception and recover; however, normal exception propagation permits compiler-specified processing (e.g., reclamation of subheaps) to be performed as each routine exits.

The effect of ABORT_TASK depends upon the state of the task at the time of termination; this state is maintained by the task management package in a field of each task control block. The tasks states and the associated abnormal termination processing is described below; in each case, any dependent tasks must also be aborted.

1. Terminated -- No processing is required.

2. Waiting at END statement -- Abort all dependent tasks.

3. Executing but not in rendezvous -- The task is terminated.

4. Executing in rendezvous -- The exception TASKING_ERROR is raised in any task that is engaged in or waiting for rendezvous with the aborted task.

5. Waiting for rendezvous to complete -- If rendezvous has not begun, the aborted task is removed from the queue of the entry it called; a timed entry call is canceled. If rendezvous is in progress, the task that made the entry call (the aborted task) is terminated but the task that is performing the rendezvous is permitted to complete; no exception is raised.

6. Waiting in ACCEPT statement -- The exception TASKING_ERROR is raised in any task that is waiting for rendezvous with the aborted task.

7. Waiting in SELECT statement -- The exception TASKING_ERROR is raised in any task that is waiting for rendezvous with the aborted task; any DELAY request is canceled.

8. Waiting in DELAY statement -- The delay request is canceled, and the task is terminated.

3.2.3.2.13 FAILURE Exception

A statement of the form

```
raise TASK_NAME'FAILURE;
```

causes the exception FAILURE to be raised in the specified task. This statement is translated into a call to procedure RAISE_FAILURE, which has as its parameter the access value for the task control block of the specified task. The processing performed by RAISE_FAILURE depends upon the state of the specified task.

1. Terminated -- If the task has already terminated, no processing is required.

2. Waiting at END statement -- No processing is required.

3. Executing but not in rendezvous -- The exception FAILURE is raised at the current point of execution.

4. Executing in rendezvous -- If the point of execution is within a block that handles the exception, the rendezvous is not disturbed. If the exception is not handled within the ACCEPT statement, the rendezvous is terminated, exception TASKING_ERROR is raised in the calling task, and the FAILURE exception is propagated to the enclosing block.

5. Waiting for rendezvous to complete -- If rendezvous has not begun, the task issuing the entry call is removed from the entry queue; a timed entry call is canceled. If rendezvous is in progress, the rendezvous is permitted to complete, and the exception FAILURE is raised in the task that made the entry call.

6. Waiting in ACCEPT statement -- The exception TASKING_ERROR is raised in any task that is waiting for rendezvous with the failed task.

7. Waiting in SELECT statement -- The exception TASKING_ERROR is raised in any task that is waiting for rendezvous with the failed task; any DELAY request is canceled.

8. Waiting in DELAY statement -- The delay request is canceled, and the FAILURE exception is raised.

3.2.3.2.14 Clock Manager

The package CALENDAR [DoD80B, 9.6] is a predefined library package that defines the type TIME, function CLOCK, and operations on items of type TIME. (There is also a function CURRENT_TIME that is similar to CLOCK except its result is of type DURATION instead of TIME.) The actual maintenance of current time is provided by a task called CLOCK_MANAGER that is local to CALENDAR and has the following entries:

1. DELAY_TASK -- This entry is called by the procedures that implement the DELAY statement, either standing alone or within a SELECT statement. Its arguments are a pointer to the task control block of the task requesting the delay and the time at which

reactivation is to occur. The delay is implemented by placing an entry for the task in a time-ordered queue that is examined by CLOCK_MANAGER whenever the system time is updated; the entry contains the reactivation time, task control block pointer, and a flag indicating a task delay has been requested.

2. DELAY_ENTRY -- This entry is called by the procedure that implements a timed entry call. Its arguments are the time at which reactivation is to occur and pointers to the task control blocks of the task requesting the time entry call and the task being called. An entry is placed in the time-ordered queue of the clock manager that contains the task control block pointers, reactivation time, and a flag indicating this queue entry is for a timed entry call.

3. CANCEL_DELAY -- This entry is called to cancel a delay request for either a task or entry; its one argument is a pointer to the task control block of the task that made the request. This pointer is used to search the queue of delay requests and remove the one being canceled. If the request is for a task delay, field DELAY_TIME of the task control block is set to 0.0 to indicate that no request is pending. If the request is for a timed entry call, field ENTRY_TIME is set to 0.0.

4. INTERRUPT -- This entry is the one that receives "ticks" from the system clock. In general, it is activated by a hardware interrupt (or simulated interrupt if execution is under a host operating system) and updates the system time that is returned by functions CLOCK and CURRENT_TIME of package CALENDAR. When an interrupt occurs, the time-ordered queue of delay requests is examined to determine if any have expired.

   When a timed entry call delay has elapsed, field ENTRY_TIME of the control block of the calling task is examined; if it is 0.0, the task is no longer waiting on the delay, and no further processing is required. If the field is not 0.0, a P operation is performed on the MUTEX semaphore of the called task to get exclusive access to its task data structures. If ENTRY_TIME of the calling task is now 0.0, exclusion is released, and no further processing is required. Otherwise, the entry call has timed out, which is indicated to the calling task by setting its ENTRY_TIME field to 0.0. The SELECTED_ENTRY field of the calling task is used to find the entry descriptor from which the calling task is dequeued. V operations are applied to the MUTEX semaphore of the called task to relinquish exclusive access and to the SUSPEND semaphore of the calling task to reactivate it.

   When a task delay has elapsed, a P operation is performed on the MUTEX semaphore of the task to get exclusive access to the task data structures. If field DELAY_TIME of the task control block is 0.0, the task is no longer waiting on the delay, and no further processing is required. If DELAY_TIME is not 0.0, the DELAY alternative of the task must be selected using steps similar to those in CONDITIONAL_ENTRY_CALL:

a. All entries are deleted from the list OPEN_ENTRY_LIST.

b. The field SELECTED_ENTRY of the task control block is set to point to the entry descriptor for its DELAY alternative so the task can determine which entry caused reactivation.

c. The field DELAY_TIME is set to 0.0 to indicate that the delay has expired.

d. The task is reactivated by a V operation applied to its SUSPEND semaphore; mutual exclusion is released by the reactivated task.

3.2.3.3 Outputs

The outputs of this package are the modifications that are made to the task data structures as described above.

3.2.3.4 Special Requirements

3.2.3.4.1 Semaphores

The implementation of task management described above assumes the existence of the abstraction SEMAPHORE with operations P ("wait") and V ("signal") [DIJ68]. It is the primitive tool that is used within the ASE to provide the synchronization required to implement the Ada tasking constructs. Semaphores will have to be implemented for each target machine; their operators will probably have to be coded in machine language to get to the machine-particular features with which lockout can be implemented. Most of the code that implements the Ada scheduling policy will occur within the semaphore operators since they are the mechanism by which a task is suspended and reactivated (except when preempted by an interrupt). The semaphore operations P and V are prime candidates for inline translation. For many target machines, it will be feasible to expand inline the code that checks the state of a semaphore and go out-of-line only if a scheduling decision must be made; e.g., if a task must suspend itself on a P operation or if a task is activated on a V operation.

3.2.3.4.2 Time Services

Machine-particular routines in package CALENDAR must be supplied with which to access the system clock of each target machine.

3.2.4 Program Management

A program is the most general entity that is available in the Ada language. It consists of one or more tasks, each of which is a separate site of concurrent execution and shares its address space with the others. For an application that executes on an embedded system, all software is structured within a single program. The situation is more complex within the Ada Integrated Environment: it is meaningful for one user to have several programs executing concurrently and for there to be several users active at one time. For example, it must be possible to use the editor to prepare one module while another is being compiled. Program management is the component of the Ada Software Environment that supports program execution on a host machine; its major capabilities are:

* interface one or more Ada programs to the host operating system and provide the capability for one program to invoke, pass parameters to, and communicate with another program;
* provide a executive program that serves as a (virtual) operator's console from which the user may enter a command language that controls the execution of his programs;
* permit the physical terminal to be organized into one or more "windows" and permit these windows to be connected dynamically to the virtual terminals that are associated with the user's programs.

This section is presents the functionality and structural relationships within the program management package. The details of the executive control language and user command language are presented in Appendices A and B, respectively.

3.2.4.1 Inputs

The program management package receives input from the following sources:

1. User's terminal -- Commands to control program execution are entered at the user's terminal and processed by the executive control and command language interpreter tasks of the program management package.

2. APSE database -- The database is consulted to verify the access rights of a user and retrieve his permanent program environment.

3. Program binder -- Information stored in the database by the program binder is used to load a program and pass parameters to it.

4. Program Debugger -- The debugger controls and examines the state of a program by sending messages to its KAPSE interface task, which returns the requested data via the inter-program communication.

5. User programs -- Messages are received that reflect its change of state of a user program.

Figure 3-38 illustrates this data flow when the user is controlling the execution of two programs from his terminal. Program A has been started and disconnected from the terminal. Program B is receiving interactive commands from the user. Appendix E contains a series of figures that illustrate a complete session at a terminal.

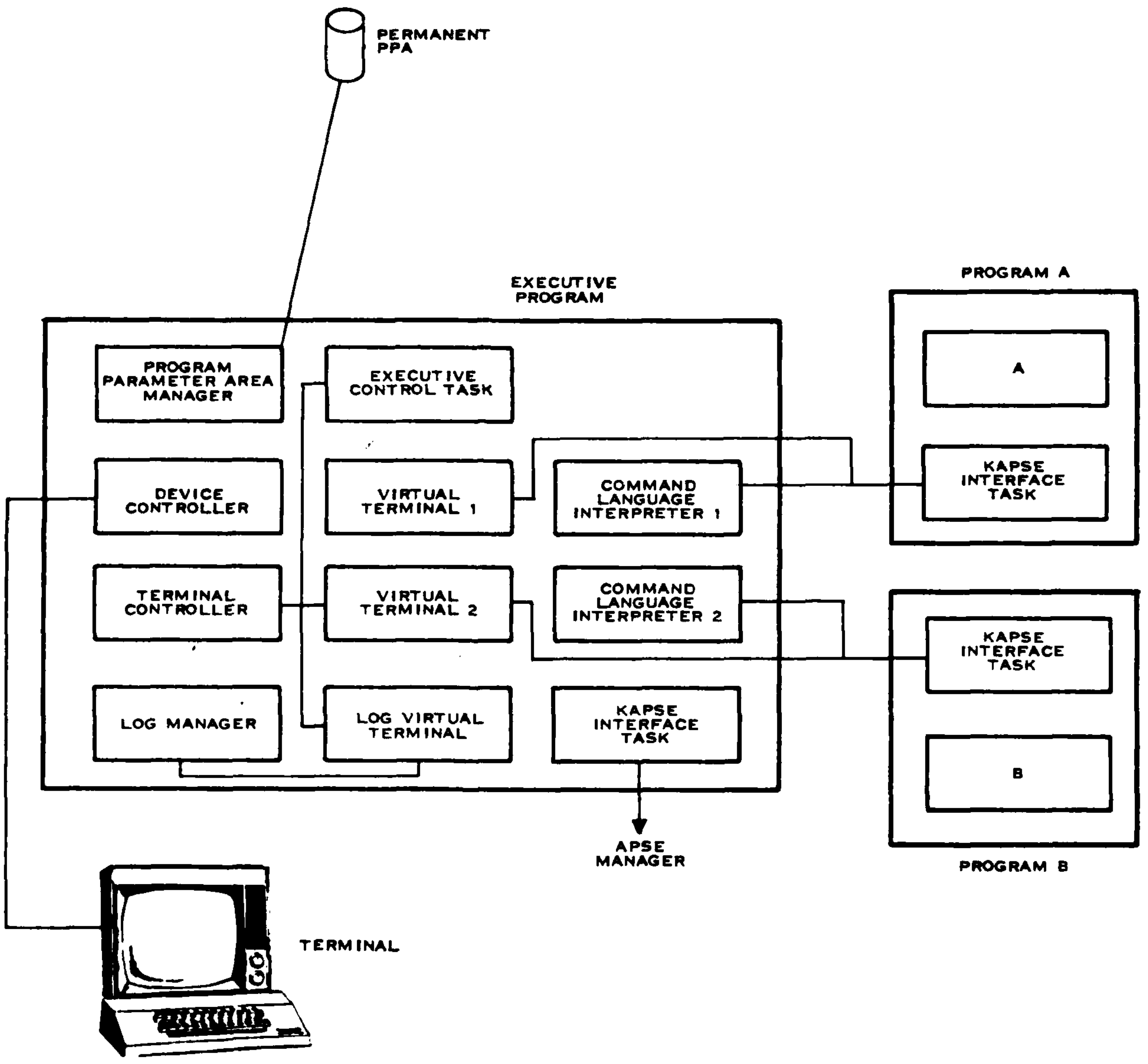


Figure 3-38 Program Management Data Flow

3.2.4.2 Processing

3.2.4.2.1 Executive Program

The executive program is an Ada program that provides the interface among a user of the Ada Integrated Environment, his terminal, the programs that he executes, and the host operating system. This interface appears to the user to be an operator's console through which he can enter two types of commands. The most primitive commands are analogous to the control language that a system operator uses to affect to flow of jobs executing under a general purpose operating system. In this case, the user's terminal is viewed as the console for the collection of programs that the user has initiated. The executive command language permits the user to observe and modify the status of his programs, control their connection to the physical terminal, and adjust the characteristics of the terminal. The second command language is used to specify the execution environment of individual programs. It is a Ada-like language with which the user can create command procedures and files that invoke one or more programs. Program parameters and command language variables are supported.

The executive program is an Ada program that has eight distinct tasks declared within it. These tasks are described in the following sections.

3.2.4.2.1.1 Device Controller

This task provides the device-dependent processing required to interface a particular terminal to the terminal controller. This interface is device-independent, at least within a broad class of device such as low-speed teletypewriter or high-speed video display terminal; it is low-level -- essentially get-a-character, put-a-character (or record) with appropriate control functions. The device controller communicates to its terminal device through either host system service calls or machine level interrupts. Within this task, transformations can be made to the standard Ada character set.

3.2.4.2.1.2 Terminal Controller

The terminal controller task uses the low-level interface that is provided by the device controller to interface the user's terminal to one or more virtual terminals; the coding for this task is machine and device independent. The screen of the actual terminal is logically partitioned into one or more windows, each of which appears to be a separate display device (cf. [WAR79]); the terminal controller provides the synchronization that permits concurrent output requests to be displayed on a single screen. The keyboard of the terminal is associated with the window in which the cursor occurs. (A suitable interpretation will be provided for terminals that have printers instead of screens and cursors.) Executive command language statements are available with which to adjust the characteristics of windows and to control the connection of a window to a program.

Two modes of access to a window are supported. In the video display terminal (VDT) mode, it is assumed that the physical terminal supports cursor-positioning and can accept data at a high enough data rate that it is feasible to rewrite a complete screen. In the teletypewriter (TTY) mode, transfers are made at the logical record level, and the "screen" is automatically scrolled up as lines are output. TTY mode is the default for a text file since this mode is available on all terminals and it requires the least knowledge of the window to which output is being sent.

An important function of the terminal controller is the processing associated with special control characters that are recognized by the device controller. In particular, it is assumed that any host machine (and terminal) on which the ASE is executing has some form of "break" key with which the user can asynchronously signal that execution should be interrupted and the user be given an opportunity to enter new commands. The terminal controller implements the following semantics for the break key:

1. Note the window in which the cursor is positioned when the break occurs.

2. If the window is connected to the executive control task, no further processing is required; execution continues.

3. If the window is not connected to any task, connect it to the executive control task.

4. If the window is connected to the command language interpreter task or a program, break the connection and connect the window to the executive control task; the disconnected task or program is placed at the head of a queue associated with the window.

The effect of this sequence is to connect a window to the executive control task, which then prompts for input in the executive control language. The disconnected task or program is not halted automatically; it continues to execute until it is halted by specific executive command or by an I/O request to its virtual terminal that blocks due to disconnection from the terminal controller.

3.2.4.2.1.3 Virtual Terminal Controller

When a program performs I/O operations to a file associated with a interactive terminal, an instance of the virtual terminal controller task serves as an intermediary that provides a simulated keyboard and screen to the program. Actual I/O to the physical terminal occurs only when the virtual terminal is connected to a window (via the terminal controller). A request for keyboard input suspends the program until its virtual terminal is connected to a window and data have been entered. An output request to the screen of the virtual terminal can normally be processed immediately since only the buffer of the virtual terminal must be updated. A data transfer to the actual screen occurs only when a window is connected. (It is possible for data to be lost if the terminal is not connected to a window; this is analogous to losing data on a conventional terminal because the operator is not watching the screen when output occurs. Executive commands will be provided with which to specify the number of output requests to be accepted by a virtual terminal before the associated program is blocked to await acknowledgment of receipt by the operator.)

Since the connection of a virtual terminal to the actual terminal is under control of the user ("operator"), there is no need to distinquish between "foreground" and "background" execution of a program. Consider the execution of the Ada compiler. The user may invoke it initially with connection to the actual terminal. After the name of the unit to be compiled has been entered and processing has begun, the user can disconnect the compiler's virtual terminal from its window and invoke another interactive program. At any time, the user can use the break key to interrupt the interactive program and switch to the compiler's virtual terminal to observe the progress of the compilation. The user can also observe the status of the compiler to determine if it is blocked while waiting for terminal I/O. (An example of this type of activity is given in Appendix E.)

3.2.4.2.1.4 Executive Control Task

When this task is connected to the user's terminal via the terminal controller, it provides a simulated operator's console from which the user can control the execution of programs he has invoked and the connection to their virtual terminals to the windows within the actual terminal. The executive interprets a simple command language (Appendix A) that provides the following functions:

1. login / logout;
2. partition of physical screen into windows;
3. connect windows to specific programs;
4. connect the terminal keyboard to a specific window.
5. display program status;
6. interrupt execution of a program;
7. halt, resume, or terminate a program;
8. instantiate command language interpreter.

3.2.4.2.1.5 Log Manager

This task serves as the recipient of messages that have been sent to the user. These messages may have originated in programs that the user invoked or in programs that are executing in the environment of other users. The log manager writes all messages to a disk file that is maintained for the user by the database. The user can also connect a virtual terminal to the log manager and specify classes of messages to be copied to the terminal; he thus receives asynchronous notification of significant events. The virtual terminal of the log manager will typically be connected to a one-line window and be parameterized to require acknowledgment by the user. Programs that are executing without connection to a window can signal completion by sending a message to the log.

3.2.4.2.1.6 Program Parameter Area Manager

The ASE must support the invocation of one program by another. It is essential that a mechanism be defined by which an Ada-coded program may be parameterized at execution time so it can determine the entities upon which it is to operate. This capability is supported through an abstraction called a program parameter area (PPA). Each program executing on a host system has an associated PPA which appears to be an unbounded storage area containing entries called synonyms. Each synonym has two string components: the name of the synonym and the value associated with it. (It is important that the PPA be implemented as a abstraction since, depending on the host machine, it may be implemented via memory, disk files, or a distributed database.) The PPA is an area into the invoker of a program may store parameters for retrieval by the program.

At any instant, there is a tree of PPAs that is accessed under the control of the PPA manager task of the executive program. This tree has a PPA for each program or command procedure that is active within the user's environment. The PPA manager provides the interface through which concurrent requests are made to access PPAs.

The root of this tree is a PPA that is permanently associated with a user; it is retrieved from the database when the user logs in and saved when he logs out. The permanent PPA is the environment of a user that persists from one session to the next; it typically contains abreviations for user-specific characteristics and file pathname components. The following points describe the semantics of the tree of PPAs.

1. An invocation of the executive control task uses the root PPA.

2. An invocation of the command language interpreter within a window of user's terminal uses the root PPA.

3. A file of command language statements is interpreted by the command language interpreter using the PPA in effect at the point input is switched to the command file; i.e., switching to a command file is equivalent to entering the commands contained within the file.

4. An Ada program is executed with connection to a PPA that is a copy of the PPA that was active in the environment in which the program was invoked; within the tree of PPAs, the new PPA is a descendant of the PPA of the program's creator.

5. A command language procedure is treated as if it were a Ada program; i.e., a command procedure is interpreted within a PPA that is both a copy and descendant of the PPA that was active when the procedure was invoked.

Thus, each level in the tree corresponds to a level of program or command procedure invocation, and each PPA is initialized by a copy of its ancestor in the tree.

The PPA provides one mechanism by which to pass parameters to a program. This mechanism is implicit in the sense that each program inherits a copy of the environment of its parent; it is not apparent textually which synonyms within a PPA will be used as parameters. (An explicit form of parameterization is provided by program arguments, which are discussed in the section on the command language interpreter.) A program is not permitted to modify the PPA of an ancestor since concurrent execution of programs could lead to unexpected changes to a program's environment. The PPA interface package shown in Figure 3-39 is provided for a program to access its PPA. This package uses the variable length string type TEXT that is defined in package TEXT_HANDLER [DoD80B, 7.6]. (A synonym is deleted from the PPA by assigning the empty string as its value; the empty string is returned as the value of a synonym that is not present in the PPA.) As an example usage of this package, suppose a program must list a file that is a member of a specific library. If synonyms LIBRARY and MEMBER specify the file and if the synonym FILE_NAME must be constructed as a parameter for the listing program, the following statements could be used:

```
GET_PARAMETER( TO_TEXT( "LIBRARY" ), LIBRARY_NAME );
GET_PARAMETER( TO_TEXT( "MEMBER" ), MEMBER_NAME );
PUT_PARAMETER( TO_TEXT( "FILE_NAME" ),
               LIBRARY_NAME & '.' & MEMBER_NAME );
```

where LIBRARY_NAME, MEMBER_NAME, and FILE_NAME are of type TEXT and pathname is constructed by appending a "." and the member name to the library pathname. After these statements have been executed, the PPA of the active program contains a synonym FILE_NAME with the desired file name as its value. If the file listing program is invoked from the active program, the file listing program can use GET_PARAMETER to retrieve its parameter FILE_NAME.

```
with TEXT_HANDLER; use TEXT_HANDLER;

package PPA_HANDLER is

procedure GET_PARAMETER( NAME  : TEXT;
                         VALUE : TEXT );

procedure PUT_PARAMETER( NAME  : TEXT;
                         VALUE : TEXT );

procedure PUT_PARAMETER( NAME  : TEXT;
                         VALUE : STRING );

procedure PUT_PARAMETER( NAME  : TEXT;
                         VALUE : CHARACTER );

end PPA_HANDLER;
```

Figure 3-39 Package PPA_HANDLER

3.2.4.2.1.7 Command Language Interpreter

The command language interpreter (CLI) is a task within the executive program that is instantiated to translate and interpret the command language with which the AIE user specifies the execution of Ada programs. This command language, which is described in detail in Appendix B, has an Ada-like syntax, uses a procedural notation to invoke programs (with arguments), and has variables for storage of program arguments and intermediate computations. (CLI variables are stored as PPA synonyms; as a result, they may be used as implicit program parameters.)

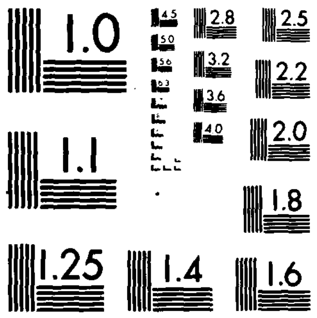
CLI is a task for which an access type is defined; a copy of CLI is instantiated within the executive program for each program that is active in the user's environment. The executive control task uses the collection of CLI tasks to control a user's programs. Since each instance of CLI is in communication with its associated program, the executive can examine and modify the state of a program by means of messages sent via CLI.

The most common method for activating CLI is through an executive command that disconnects a window from the executive control task and connects it to an instance of CLI. Text entered at the terminal is considered by CLI on a line-by-line basis. If text is recognized as a CLI primitive command, it is executed directly by CLI. Examples are assignments to command language variables and structured control statements. A valid CLI command that is not a primitive must have the syntax of an Ada procedure call. CLI searches to one or more library pathnames that have been supplied as defaults by the user to determine if one contains a file with the same name as the procedure. If a file is not found, the command is in error. If a file is found, the processing that is performed depends upon the characteristics of

AD-A109 976 TEXAS INSTRUMENTS INC LEWISVILLE EQUIPMENT GROUP F/G 9/2
ADA INTEGRATED ENVIRONMENT III COMPUTER PROGRAM DEVELOPMENT SPE--ETC(U)
DEC 81 F30602-80-C-0293
UNCLASSIFIED RADC-TR-81-360-VOL-1 NL

2 of 2
AD
A109976

END
DATE
FILMED
3-82
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

that file.

1. If the file contains a program, the program is executed under the control of the current CLI. The arguments of the program call are verified with the argument types that were declared for the program, and the arguments are converted to their internal representation. A copy of the active PPA is created for use by the new program. A program management primitive operation is used to start execution of a skeletal program that contains the KAPSE interface task (KIT), which in turn loads the program image for the program that is to be invoked. The program arguments are transmitted to KIT, who uses them to initialize the stack frame of the program and activates to user's program. CLI suspends itself to await the completion of the program or a request by the executive control task to query or modify the state of the program. When the program terminates, CLI receives a message from KIT and retrieves any OUT program arguments, which are converted to string form and stored in the original PPA. If the program terminates due to an exception and CLI is executing from a virtual terminal, the data structures of the program are not deleted until the user has an opportunity to activate the debugger and examine the cause of abnormal termination.

2. If the file is a command procedure (a command language file that is structured as a procedure (possibly with arguments) and has been saved in a pre-processed form by CLI), its effect is comparable to that of a program except its statements are interpreted by CLI instead of being executed as a distinct entity of the host operating system. The arguments of the procedure call are verified with the argument types that were declared for the procedure, and the arguments are stored into a copy of the active PPA. CLI then interprets the text of the procedure within the context of the new PPA. (This text will ultimately result in the execution of command language primitives or programs.) Any OUT arguments of the procedure are stored into the original PPA when the procedure terminates.

3. If the file contains text, CLI switches its input to the file and attempts to interpret it as a script of command language statements. When interpretation reaches the end of the file, CLI resumes reading the original file for its input.

4. If the file is not one of the preceding types, the command is erroneous.

A second method by CLI is invoked is through the command invocation package (Figure 3-40) that supports the invocation of a CLI command from within a program. Function INTERPRET causes a copy of CLI to be instantiated to interpret the specified command text. The identifier that is returned by INTERPRET may be used to monitor (STATUS) and control (HALT, RESUME, TERMINATE) the execution of the command. The command is interpreted by CLI within the PPA of the program from which CLI was invoked. If the command is a program or command procedure, any OUT arguments can be examined in current PPA.

```
with TEXT_HANDLER; use TEXT_HANDLER;

package PROGRAM_INVOCATION is

  type COMMAND_ID is limited private;

  type COMMAND_STATUS is ( "some enumeration" );

  procedure HALT( ID : COMMAND_ID );

  function INTERPRET( COMMAND : TEXT )
    return COMMAND_ID;

  procedure RESUME( ID : COMMAND_ID );

  function STATUS( ID : COMMAND_ID )
    return COMMAND_STATUS;

  procedure TERMINATE( ID : COMMAND_ID );

private

  type COMMAND_ID is "to be determined";

end COMMAND_INVOCATION;
```

Figure 3-40 Command Invocation Package

3.2.4.2.1.8 KAPSE Interface Task

Each Ada program that executes on a host machine has within its address space an instance of the KAPSE interface task (KIT). As its name implies, this task serves as the intermediary between the tasks that comprise the Ada program and the environment that is external to the program. KIT's primary functions are to load the user's program and to communicate with the command language interpreter and with other programs.

A request from CLI to execute a program causes a series of actions to occur. A host operating system service request is issued by CLI that causes the host operating system to load a canonical Ada program. The code space of this program contains only KIT and the ASE support routines that are shared by all Ada programs; the data space of this program is configured as a heap. (If the host operating system does not permit the address space of a program to be expanded dynamically, versions of the canonical program will be provided that support a range of program address spaces.) The canonical program is established with a low-level I/O link to the instance of CLI from which it was invoked. Over this link, CLI sends a message to KIT that specifies the pathname of a program file and the mode in which it is to be loaded. KIT uses the program heap to load the program, build any linkage tables that are required by the type of load, and allocate the stack region for the task associated with the program module. After the program has been loaded, CLI sends KIT the binary form of any program arguments, and KIT stores these arguments in the stack frame of the program module. The program is treated as an anonymous task for which a task control block is constructed. KIT initializes the saved context in the control block so execution of the program will begin in a routine that elaborates the global packages used within the program; the packages and the order in which they are elaborated are specified in a list that was prepared by the binder. KIT also arranges for the initialization of the standard text input and output files. KIT concludes the program load by placing the task control block for the program in the scheduling queue and suspending itself to await either termination of the program, a primitive I/O request from the program, or a message from CLI. The priority of KIT is higher than that of any user-defined task; this ensures that requests from CLI are treated as interrupts and are honored immediately.

Based on the type of program binding that was selected, KIT supports the following modes for loading.

1. Memory image -- A program in this format is essentially complete. Its segments have been bound into a single entity containing inter-segment reference tables that are complete except for shared ASE segments. This type of loading is the fastest since it requires only that the image be read into the heap and that references to shared ASE routines be resolved. Since the relative locations of the program segments were chosen by the binder, this format in practice requires hardware support for mapping contiguous logical addresses into discontiguous physical address space; otherwise, a program cannot be loaded unless a single contiguous block of memory can be found.

2. Dynamic linking of segments -- In this type of loading, inter-segment reference tables are created by KIT as the program is loaded. The binder prepares a list of segments that comprise the program and specifies the program entry point. KIT loads each segment into the heap and uses the definition and reference tables at the head of each segment to build the associated dictionaries, which requires resolution of inter-segment references. Since segments are processed individually, there are fewer problems finding contiguous blocks of memory, and it is easier to share segments among programs.

   The primary motivation for supporting this type of binding is that it permits a single segment to be shared by several programs without forcing global constraints upon the nomenclature that is used for inter-segment reference. Each program has its own segment table and dictionaries, so a given segment may have different segment numbers in different programs. Thus, a segment may be used freely in a program without concern about the logical address at which it is loaded or how it is referenced in programs that use it concurrently.

On an embedded system, the analogue of this type of loading is the construction of inter-segment reference tables at system initialization time. It is feasible for the target system run-time support system to contain a (small) initialization routine that builds the inter-segment reference tables based upon a list of addresses at which segments may occur, typically in read-only memory (ROM). Usage of this technique means a new version of a package can be installed by inserting a new ROM; the ROMs containing code that references this package are not made obsolete. (This support for modular software has been demonstrated in Texas Instruments Microprocessor Pascal [BAT79, TI81D].)

3. Overlaid image -- The user may specify to the program binder that the segments comprising a program be grouped into overlays that have user-designed memory residence relationships. The actual loading of overlays is performed automatically by a modified version of the Ada linkage handler. Recursive subprogram calls between overlays and calls to parallel overlays are supported.

4. Demand paging of segments -- For this type of loading, KIT manages a region of memory into which code segments are loaded as they are referenced. This is similar to overlay loading except the logical address at which a segment is loaded is not fixed. Demand paging typically has more overhead than overlaying but is easier to use since it does not require analysis of inter-segment reference relationships and segment sizes. Demand paging is well-suited to the development stages of a program; overlaying can be used in the production version.

   On an embedded system, the analogue of demand paging is automatic management of page registers to permit a large program code space to be accessed through a small logical address space. For example, the military standard 1750A instruction set architecture [DoD80C] supports a 16-bit logical address space for the instructions of a program but provides sixteen mapping registers through which 4096-word blocks are mapped into a 20-bit physical address space. A special version of the Ada linkage handler can be used to adjust mapping registers at subprogram call and return in such a manner that all the instructions for a program are accessed through one or more 4096-word windows in logical address space. (On some embedded systems it may be appropriate to have the linkage handler manipulate ROM enablement lines to select code segments.)

The code within KIT that loads a memory image or dynamically links segments is not needed once program execution begins. For host systems that have a limited address space, the memory that is occupied by this code can be incorporated into the heap of the new program and re-used.

The final function of KIT is to provide the asynchronous communication interface that is needed for inter-program communication. KIT has the structure of an interrupt handler: one or more entries are called as pseudo-interrupts whenever a message arrives from another program. KIT decodes the message and performs the required processing. This technique is used for all inter-program I/O and for control of a program by the command language interpreter or debugger.

3.2.4.2.2 VM/370 Program Management

The standard program management package described in the previous section assumes that an underlying host operating system manages the resources of individual programs and schedules their execution. Under VM/370, each user of the AIE is given his own virtual machine in which his programs execute. In this environment, the program management component of the ASE must be extended to provide the services of a host operating system. In particular, the ASE must manage the address space of each program, schedule programs for execution on the user's virtual machine, and convert interrupts at the virtual machine level (such as expiration of the interval timer and arrival of inter-machine messages) into simulated interrupts at the program level.

3.2.4.2.3 Application Programs

An Ada program that executes on a host system is an executable entity of the host operating system and contains within it the KAPSE interface task (KIT) and shared ASE support code. As was described above, KIT loads the executable code for the program and provides an interface for communication with external programs. In particular, it is through KIT that a user may control execution of a program from his terminal or from the debugger.

The tasking that is defined in the Ada language is not implemented by having each task be a separate schedulable entity with respect to the host operating system. Many operating systems restrict the number of sites of execution that they support and do not provide the complex sharing of data space that is possible with Ada tasks. The ASE implements tasking within a program by providing a separate context for each task and using the Ada priority rules to decide which task should be active.

On an embedded system, a program will typically contain only the ASE support code that is required to support its algorithms; the program is self-contained and does not require the inter-program support that is provided by KIT. However, it may be convenient to have a simple version of KIT present to provide communication if a system is to be loaded or debugged from a remote computer.

3.2.4.3 Outputs

The program management package produces the following output (Figure 3-38):

1. User's terminal -- The executive control task and command language interpreter display information on the user's terminal.

2. APSE database -- The database receives log data for a session and

any modifications to the user's permanent program environment.

3. Program Debugger -- The debugger examines the state of a program by sending messages to its KAPSE interface task, which returns the requested data via the inter-program communication package.

4. User programs -- The executive program sends messages to a program to control its execution.

3.2.4.4 Special Requirements

Implementation of the program management component of the ASE requires that the host operating system provide the following capabilities.

1. Asynchronous entry from keyboard -- It must be possible for a user to enter some keystroke sequence from his terminal that signals an "interrupt" to the program connected to the terminal. That is, it must be possible to force a program to save its current context and wait for directives to be entered from the terminal. This will be possible if there is some form of keyboard input request that causes a context switch when data are entered. (An asynchronous interrupt ensures that the user can always regain control of his terminal; a weaker condition is that the device controller be able to test periodically for a particular key. This requires that processing within the executive program be structured in such a manner that the device controller is not locked out of execution for "too" long.)

2. Asynchronous inter-program messages -- There must be some technique for passing data from one program to another that causes a context switch in the receiving program. With this capability, a general inter-program communication package can be implemented. An interrupt upon data arrival is necessary so a program such as the executive or the debugger can control another.

3. Dynamic program execution -- It must be possible for one program to place another in execution and receive the identity of the new program so messages can be passed to it. The program that is loaded may be in any format that is convenient for the host operating system since the same image is loaded for all Ada programs. This canonical program contains KIT and enough ASE support routines to load an Ada program in each of the four modes that is supported by the binder.

3.2.5 APSE Manager

3.2.5.1 APSE Manager

The APSE manager is a special usage of the program management component of KAPSE to execute a collection of programs whose existence is associated with the Ada Integrated Environment, not a particular user. The instance of the executive program that controls the APSE manager's environment has a special terminal interface that permits execution in a mode without connection to a physical terminal (and user). Examples of programs within the APSE manager are the database, batch monitor, and data communication handler.

3.2.5.2 Inputs

The APSE manager is a pseudo-user that receives input from its terminal only when connected for debugging. Programs that execute within the APSE manager receive their input through inter-program communication paths.

3.2.5.3 Processing

Execution of the Ada Integrated Environment begins by using host system services to start execution of an instance of the executive program (without a login). Within this program is a special version of the device controller task that communicates with a simulated terminal instead of a physical terminal and user. The data read from the simulated keyboard come from a host system file and consist of a script of executive and command language statements that place into execution the component programs of the APSE manager. After this "bootstrap" is complete, the device controller goes into a mode in which it waits for commands from an inter-program communication file. If it becomes necessary to examine the environment of the APSE manager, a system programmer can execute a program that connects his terminal to the APSE manager through an inter-program file.

The following programs will reside within the APSE manager.

1. Database Manager -- This program implements the database of the Ada Integrated Environment.

2. Accounting Manager -- This program handler login/logout and accounting for resources used by program and user.

3. Batch Monitor -- This program controls the execution of batch streams, sequences of command language statements that are executed without connection to a specific terminal. A user requests batch execution by sending a message to the batch monitor that contains the name of the command file to be interpreted and the file on which the command listing is to be made. Execution occurs under the control of an instance of the command language interpreter that is activated by the batch monitor.

4. Data Communications Handler -- This program provides communication with other machines. Access to the ARPA net will be provided.

5. Spooler -- It may be convenient to control spooling of listing

files within the AIE so output from a session can be logically grouped.

3.2.5.4 Outputs

The APSE manager is a pseudo-user that outputs data to its terminal only when connected for debugging. Programs that execute within the APSE manager produce output on inter-program communication paths in response to service requests.

3.2.5.5 Special Requirements

Careful attention must be paid to the bootstrap of the APSE manager. In particular, the database manager may have to be loaded using host system services since the normal KAPSE loader uses output of the binder that has been stored in the database.

3.2.6 Input/Output

The input/output (I/O) component of the ASE provides an interface for communication from an Ada program to a user at a terminal, the AIE database, or other program. This component is part of the KAPSE virtual interface; as such it presents a device and host system independent interface to the user. The I/O component is implemented in terms of three layers of routines: high-level I/O, logical record I/O, and virtual device I/O. Each layer is implemented in terms of its successor; the calling sequences for the virtual device I/O routines define the KAPSE virtual I/O interface.

This section discusses the structure of the I/O component and how it is implemented. Details of the user interface are given in Appendix C. This design is based on the configurable, device-independent I/O system that has been implemented for Microprocessor Pascal [TI81B].

3.2.6.1 Inputs

The basic input to all routines of this component is a structure called a file descriptor that contains the data with which a file is modeled. (The descriptor that is used high-level and logical record I/O is device independent; the one that is used by device I/O contains fields that are device dependent.) Individual routines may have additional inputs.

3.2.6.2 Processing

3.2.6.2.1 High-Level I/O

An Ada programmer performs input and output using the high-level I/O routines which are provided in the packages INPUT_OUTPUT and TEXT_IO (Appendix C). These packages are modifications of the packages with the same names in [DoD80B]. The package INPUT_OUTPUT is generic and must be instantiated for each element type for which file I/O is to be performed.

TEXT_IO is a derivative of INPUT_OUTPUT that operates on files whose elements are of type CHARACTER; a substructure of lines is maintained within text files.

The primary characteristic of these packages is that they have a user-oriented interface. Operations are performed in terms of a file abstraction that is at high enough a level to hide media-specific characteristics. Compile-time checking of operations is possible since the packages are generic with respect to file element type and files are distinguished by mode of transfer: read (IN_FILE), write (OUT_FILE), or read-write (INOUT_FILE).

The high-level I/O packages are tansportable since they are device and media independent with respect to user interfaces and implementation. Type-dependent processing is isolated in the high-level I/O packages so all instantiations can use a single logical record I/O package.

3.2.6.2.2 Logical Record I/O

The logical record I/O package has essentially the same interface as the package INPUT_OUTPUT except it is not generic. The same functions are performed as in INPUT_OUTPUT but with canonical parameter types; that is, requests for I/O are made in terms of the address of a high-level file descriptor, the address of a buffer, and the number of storage units of data to be transferred. With respect to its interface and implementation, this package is device and media independent. Actual I/O operations are performed through the virtual device I/O package, whose calling sequences form the KAPSE virtual interface for I/O.

3.2.6.2.3 Virtual Device I/O

The virtual device I/O component of ASE I/O is a collection of packages, one for each type of (virtual) device that is supported. Examples are terminals, printers, modems, inter-program communication channels, the database file system, and the host file system. The KAPSE virtual I/O interface is defined by the visible part that is common to specification of each of these packages; each package body is the implementation of that interface for a particular device type. (The actual machine and device dependent operations that perform I/O will be defined in each device package, not in a single package with overloaded subprograms [DoD80B]; this approach preserves the configurability of virtual device I/O packages.)

Since the association of a high-level file with a particular device can be based on a pathname that is computed during execution, the calls to the virtual device I/O package that occur in the logical record I/O routines must be parameterized so a particular device can be selected dynamically. Moreover, this parameterization should be implemented in such a manner that a new device can be added incrementally; that is, adding a new device does not require any portion of the I/O component of the ASE to be recompiled.

The parameterization of device services is provided by having each device package build a device service vector during its elaboration. This vector is an array of addresses, one for each of the subprograms in the visible part of the virtual device package. A request at the level of the KAPSE virtual interface is made indirectly through a canonical service routine that has a device service vector as a parameter and uses that vector to invoke the corresponding subprogram of the virtual device.

The association of a high-level file to a particular device occurs in following way. At program initialization, each of the device interface packages that is available for use in a program is elaborated and "registers" itself with the ASE by enqueuing its service vector in a list of all available devices. After a high-level file has been initialized, the subprogram SET_NAME is used to specify the string that becomes the external name attribute of the file. Subprograms CREATE and OPEN use this attribute to establish an association between an internal file and a device (or disk file). Connection is made through a subprogram CONNECT that is provided in the virtual device I/O component. For each device package that has been elaborated, this subprogram calls the associated CONNECT service handler to test the external file name. If the name is recognized by the package, CONNECT returns a low-level file descriptor that contains both device-specific parameters and a pointer to the package's service vector. A pointer to this descriptor is stored in the high-level file descriptor so the service vector associated with a high-level file can be found when subsequent high-level I/O requests are made.

The following device packages will be supplied:

1. Terminal -- A package is supplied for each type of terminal that is supported. Remote users are supported by treating a modem as a terminal.

2. Printer -- Data that are sent to the printer are assembled on disk until a logical file is complete and then spooled to the actual printer.

3. Database files -- Database nomenclature is used to connect a high-level file with a database object.

4. Host files -- The nomenclature of the host file system is recognized.

5. Inter-program communication -- This package supports connection to virtual ports through which programs can communicate. If two programs connect to the same port, the data that are produced by one program can be consumed by the other. Communication among different (physical and virtual) machines is supported.

3.2.6.3 Outputs

Each of the routines of this component of the ASE may modify data within the file descriptor that is passed as a parameter to identify the particular file or device with which communication takes place. Individual routines may have additional outputs.

3.2.6.4 Special Requirements

The canonical subprograms that use a device service vector to call a device-specific subprogram must be written in machine language since Ada does not support invoking a subprogram through a variable that contains an entry point address. (The service vector can be constructed in Ada using the attribute ADDRESS of a subprogram.)

SECTION 4

QUALITY ASSURANCE PROVISIONS

4.1 Introduction

Testing of the Ada Software Environment (ASE) will be in accordance with the schedule, procedures, and methods set forth in the following documents:

1. Contractor's Computer Program Development Plan (CPDP);
2. Computer Program Test Plan for this CPCI;
3. Computer Program Test Procedures for this CPCI.

Testing of the ASE will be performed at three levels:

1. Computer program component test and evaluation;
2. Integration test, involving all components of the CPCI;
3. Computer program acceptance testing, involving the APSE.

4.1.1 Computer Program Component Test and Evaluation

This level of testing supports development of ASE algorithms. Since the ASE provides run-time support for all Ada programs, initial development will use the bootstrap compiler and its simplified environment. When enough components of the ASE have been tested to provide a minimal execution environment, the bootstrap support package will be replaced.

Each component of the ASE will be tested in a stand-alone program before integration. This testing will concentrate on areas for which either new algorithms have been developed or there is relatively high risk. Examples of such areas are:

* Terminal handlers;
* Inter-program communication;
* Dynamic binding of program segments;
* Tasking;

* Configurable input/output;
* Virtual machine management.

Test results will be recorded in informal documentation; formal test reports are not required.

4.1.2 Integration Testing

This level of testing supports integration and prepares for acceptance tests. During this testing, ASE components will be integrated, one at a time, and run with previously tested subsets of the complete ASE. Testing at this level will follow the test plan and procedures for the CPCI. Formal test reports are not required.

4.1.3 Formal Acceptance Testing

This testing assures that the ASE conforms to the requirements in the Type A and B5 specifications. A formal test plan and test procedures will be generated and used to ensure conformance to the requirements. Acceptance tests will be defined to test incrementally the major functional components of the ASE. Acceptance testing will be documented in accordance with the Computer Program Development Plan and Computer Program Test Plans, and delivered to the Government with final system documentation.

4.2 Test Requirements

Unit testing and integration testing will be performed using the developed ASE and needed drivers. While formal test plans are not required, testing will keep the final acceptance tests in mind. Unit tests and integration tests consist of four primary phases:

* Single program, single task;
* Single program, multiple tasks;
* Multiple programs under a host operating system;
* Multiple programs on a virtual machine.

4.2.1 Single Program, Single Task

This level of support corresponds to the environment of the bootstrap compiler; it suffices to test algorithms that do not involve concurrent execution. This testing will be performed under VM/CMS on the IBM/370 and OS/32 on the Perkin-Elmer (Interdata) 8/32. The primary features to be verified are the Ada execution environment and the high-level I/O interface. Since the Ada optimizing compiler, database, and program binder will not be available when this testing is performed, host system object formats and link editors will be used. At the conclusion of this phase, an adequate environment will be available for the initial development and testing of other components of the Ada Integrated Environment.

4.2.2 Single Program, Multiple Tasks

In the next stage of testing, the ASE is upgraded to include task management, the component that supports simulated concurrent execution within a single program. This component involves complex, time-dependent algorithms that must be tested extensively.

4.2.3 Multiple Programs under a Host Operating System

This phase of testing corresponds to the addition of the program management component, whose implementation depends on the task management component that was verified in the previous phase. The primary features that must be tested are:

* dynamic program invocation;
* executive program:
  - terminal device controllers,
  - virtual terminal interface,
  - executive command language,
  - command language interpreter;
* inter-program communication.

At the conclusion of this phase, an adequate environment will be available for the initial integration testing of the components of the Ada Integrated Environment. Under OS/32, the ASE is functionally complete. On the 370, the programs associated with a user are executing under CMS in a single address space (i.e., no virtual memory management is provided by ASE); concurrent execution of programs is supported through time-slicing.

4.2.4 Multiple Programs on a Virtual Machine

The program management component is extended to become a virtual memory operating system that supports concurrent execution of multiple programs within a user's virtual machine. The design of this system is coordinated with the hardware "assists" that are provided in the 370 firmware [IBM77, MAC79].

4.2.5 Rehosting Tests

Parallel sets of tests for the first three phases will be run on the IBM 370 and the 8/32.

4.3 Acceptance Test Requirements

The acceptance tests will be run according to the contractually developed test plan. The acceptance test will consist of the four groups of tests described above.

4.3.1 Performance Requirements

The performance of the ASE will be measured in terms of its use of host system resources and the efficiency of the software service it provides.

The Government will specify the machine and operating system configurations for the initial Ada Integrated Environment host systems. Acceptance test plans will specify ASE performance requirements in terms of processing speed and memory use in these host systems.

4.4 Independent Validation and Verification

An independent validation and verification (IV&V) contractor, if one participates in the Ada Integrated Environment program, may perform independent testing of the ASE using the tests described above or special-purpose tests developed by the contractor.

APPENDIX A

EXECUTIVE COMMANDS

This appendix describes the commands which are interpreted by the executive control task of the ASE executive program. These commands provide the following functions:

1. login / logout;
2. interrupt an interactive program;
3. specify identification line for each window;
4. partition of the physical screen into windows;
5. connect windows to specific programs;
6. move between windows;
7. display program status;
8. terminate a program;
9. halt/resume a program;
10. instantiate command language interpreter.

These commands may be entered when the user is prompted by an exclamation point ("!"), except as noted below. An example terminal session in which these commands are used is in Appendix E.

A.1 LOGIN Command

Syntax: LOGIN username

The LOGIN command obtains access to the ASE. It is the first command entered by a user when the ASE executive program is connected to a user's terminal. This command verifies a user's identification and password, starts collection of accounting data, and returns the user's permanent environment.

The LOGIN command can be configured to perform installation-dependent processing (such as collection of special accounting data) by the site at which the ASE is being used; it will typically consist of a single required parameter, which is the user's identification name (username).

During processing of the command the user will be requested to enter a password for verification of access rights. On video display terminals the password will be requested in a nondisplayable field (if the terminal used has that capability). On a teleprinter the password will be requested in a field which has been overprinted by several different characters.

The LOGIN command completes its processing by either

1. informing the user that access to the ASE has been granted and returning the user's permanent environment, or

2. informing the user that access to the ASE has not been granted.

A.2 LOGOUT Command

Syntax: LOGOUT

The LOGOUT command terminates a session with the ASE. The user's environment is stored, and the accounting data is concluded.

The LOGOUT command can be configured to perform installation-dependent processing (such as final recording of accounting data) by the site at which the ASE is being used. There are no parameters for this command.

A.3 BREAK Command

This command is entered by pressing the BREAK key defined for the particular terminal that is being used. If a user-program is associated with the window in which this key is pressed, that program is disassociated from the window, and the executive program is connected. If the BREAK key is pressed when the executive program is associated with a window, the command has no effect.

A.4 HEADER Command

Syntax: HEADER ON | OFF

The HEADER command establishes or removes the header line information displayed above each window. The header line displays information concerning the window (its identification number and number of rows) and the program that is associated with the virtual terminal connected to the window (its name, identification number, and state).

The command HEADER ON indicates that the header line should be displayed for each window, and the command HEADER OFF indicates that the header line should not be displayed for any of the windows. The default for this command is HEADER ON.

A.5 SPLIT Command

Syntax: SPLIT size

The SPLIT command partitions the window from which the command is entered. "size" indicates the number of rows that the upper window should have upon completion of the command. If no parameter is specified, the window from which the command is entered is split in half. If the number of rows is odd the upper window is given one more row than the lower window.

Upon completion of the command, the executive program is associated with the upper window, and the lower window has no virtual terminal connected to it.

A.6 MERGE Command

Syntax: MERGE

The MERGE command combines the window from which the command is entered with the adjacent windows to which no virtual terminals are connected. This command results in a single window to which the executive program is connected.

A.7 ATTACH Command

Syntax: ATTACH program_id

The window from which the ATTACH command is entered is connected to the virtual terminal of the program with the specified program_id.

If no program_id is specified, a program is selected from a circular list of programs in the user's ASE which have the same size as the current window.

A.8 NEXT Command

Syntax: NEXT window_id

The NEXT command moves the cursor to the window of the specified screen. If the identification number is not specified, the selected window is the one below the window from which the command is entered. If the command is entered from the bottom window of the screen, the cursor is positioned in the top window of the screen.

There are two situations that may arise from the movement between windows.

1. If the new window is waiting for terminal input, the cursor is moved to the point at which data is to be entered.

2. If the new window is not waiting for terminal input, the program associated with the window (if any) is interrupted in the same manner as if the BREAK key were pressed.

The window from which the NEXT command is entered is returned to the state it was in before the window was interrupted.

A.9 STATUS Command

Syntax: STATUS

The STATUS command displays the status of all of the programs which are currently in the user's ASE. Information displayed by this command consists of:

* program name;
* program identification number;
* program status:
  - Waiting for terminal input
  - Running
  - Halted

A.10 KILL Command

Syntax: KILL program_id

The KILL command causes abnormal termination of the program with the specified program identification number.

A.11 HALT Command

Syntax: HALT program_id

The HALT command suspends execution of the specified program until a corresponding RESUME command is entered. This command will normally be used to "freeze" a program so that it can be examined by the debugger.

A.12 RESUME Command

Syntax: RESUME program_id

This command resumes execution of the specified program following a HALT command.

A.13 NEW_CLI or ? Command

```
Syntax: NEW_CLI
          or
         ?
```

The NEW_CLI or "?" command instantiates a new instance of the command language interpreter that is connected to the window from which the command is entered. If desired, a command to be executed by the command language interpreter may be specified by placing it immediately after the "?".

APPENDIX B

COMMAND LANGUAGE

B.1 REQUIREMENTS

The CLI (Command Language Interpreter) is a task within the executive program (EXEC) that is instantiated to translate and interpret the command language with which a user of the Ada Support Environment specifies the execution of Ada programs. The CLI can function in an interactive manner with a user and, by means of the Batch Monitor function of the Executive program, also supports Batch execution. The EXEC controls each user's virtual terminal and also controls the CLIs which are active for each user.

Each user can have one or more CLI's active at any one time during a session. There will be a CLI associated with each active user task.

The CLI has the capability of executing or causing the execution of:

1. Command Language files stored in command libraries. These are either:
   a. Command procedures which require parameters for execution and have special processing requirements before they can be executed.
   b. A sequence of command language commands.
2. Ada program files stored in the database.
3. Primitive procedures; these are inherent functions of the command interpreter.

B.2 PHILOSOPHY and DESIGN GOALS

The Command Language could be considered as the means by which a dynamically defined user program is created. As each command is issued, it is processed and then disappears. This is analagous to the repetitive creation of independently executable blocks, and leads to the conclusion that the Command Interpreter exists in 2 states: either awaiting/accepting input or acting upon that input. User input of declarations and statements can generally occur at any time during an interactive session, except where syntactically prohibited by a particular construct.

With these thoughts in mind, the following design goals are listed in priority order:

1. Provide a user friendly environment, to include:

   a. easy to remember and consistent parameter names

   b. very explanatory error messages and prompts

   c. on line help facilities with examples

2. Cater to differing user abilities and experience by providing different operational modes including:

   a. beginner mode

   b. advanced mode

   c. the capability to have different, user defined, forms of the same basic command.

3. Maintain, to the greatest degree possible, an Ada-like syntax.

B.3 CAPABILITIES and CONSTRAINTS

Following is a general discussion of the capabilities and limitations of the CLI:

1. Users shall have the ability to use the Command language like any other programming language. The Command Language will provide object declaration capabilities, looping, conditional control structures, procedure calls, etc. The full command language syntax is defined in paragraphs B.14 and B.15.

2. Users shall have the ability to create their own commands and command libraries with an explicit searching order established. Thus if a command exists in two or more libraries, the first one found is executed. The user will be able to change and override that search order.

3. Users shall have the ability to tailor their individual operating environment as regards command prompting, cursor tabs positioning and other screen functions.

4. There will be controlled access to commands based on user attributes. In this area, the CLI interfaces with services provided by the DataBase Subsystem. All files have access attributes concerning read, write and execute. Each user also has access attributes, maintained in the User Information Directory, which must be interrogated and evaluated prior to accessing any

file. The file attributes must be matched with the user, thereby providing a flexible mechanism for system security and methodological control.

Example: a user may have execution rights to the system PRINT command, have read rights to the same command, but not have write (updating) rights. On the other hand, if a user had a PRINT command defined in his own library, all access rights would be available.

5. The user will have the capability to submit BATCH jobs. A Batch job has no direct interaction with the terminal. Any attempt to access the terminal will result in immediate job termination because that device is not allocated. The user will have the ability to query job status and examine the contents of a file containing status messages and completion information regarding an individual batch job. The system will automatically inform the user of:

   * Any completed batch jobs at log-in time

   * Batch job completion while the user is still on the system.

   * Batch status at logout

6. A HELP facility shall be available both generally (if the user has no knowledge of what commands are available, or which command is applicable for a particular situation) and specifically (if the user knows the name of the desired command, but is unsure of how it works). This capability applies generally to those commands which are system defined, but may be extended by the user.

7. There is no distinction between foreground and background tasks and no need to provide a facility to specify background or foreground execution. Access to the terminal will be controlled by the Ada Software Environment through the virtual terminal concept.

B.4 OVERVIEW OF COMMAND PROCESSING

The user's primary means of interacting with the CLI is the procedure call. This is true because the procedure call is the primary means of invoking particular system functions. While CLI supports the use of if statements, loops and other constructs, accepts input and delays execution until a construct is syntactically complete, and the user may want to create a new command procedure, the procedure call remains the most fundamental command form. A procedure call is handled by the Command Interpreter with the following steps:

1. Syntactic analysis of the command

2. Search for the command in the established library search order.

3. Accessing the command file and, if applicable, the parameter data structure.

4. Parameter matching and type checking, including PPA references.

5. Merging the program and file attributes contained in a program or command procedure with those specified at execution.

6. Issuing a service request to the Program Management component of the Ada Software Environment. This request loads the canonical Ada program containing the KAPSE Interface Task (KIT). Using interprogam communication, CLI sends messages to KIT requesting loading of the program and passing program parameters.

The user can issue a procedure call in either the beginner or advanced mode. The beginner mode is signified by entering the name of the procedure, for example, PRINT_FILES. While the procedure PRINT_FILES shown in Example B-3 has four parameters, the user does not have to enter them at this time. The CLI will recognize that parameter values are missing when parameter matching and type checking is accomplished, and generates a template, shown in Example B-1, which the user must complete. In the template the user must use quotes to distinguish between string values and references to variables stored in the PPA. Default values, as indicated in the command procedure specification, will be displayed and can be altered by the user.

```
PRINT FILES
    FILE LIST?        =  (to be completed by user)
    NUMBER OF COPIES? = 1
    ANSI CONTROL?     = "Y"
    LISTING DEVICE?   =  (to be completed by user)
```

Example B-1 TEMPLATE COMMAND FORM

Once a user has developed an understanding of a command, its parameters, and any short-hand forms, the advanced forms, as shown in Example B-2, may be used.

```
PRINT_FILES (LISTING_DEVICE => "LP02");

PRINT_FILES (FILE_LIST => "ADA.CLI.SPEC,ADA.CLI.INPUT",
      LISTING_DEVICE => LP02);

PF2 ("ADA.TUTORIAL.DOCUMENT",4);
```

Example B-2 ADVANCED COMMAND FORMS

Both named (keyword) and positional parameters are supported. A file name specified in a string may have to be expanded by the Command Interpreter to its full name. This is because Synonyms (PPA entries) may be used within the string containing a file name.

When the advanced mode is used, the template will only appear if all the parameters have not been specified, or if an error has been made in specifying the actual parameters.

In the second example above, the parameter LP02 is not in quotes. As such, it would be treated as a variable, and an attempt would be made to find it in the PPA, and pass on the value associated with it. If it was not found in the PPA, and the formal parameter associated with it was specified to be of mode IN or IN OUT, the template mode would appear so that the user could input the correct data, if no default value had been provided. If a parameter is of mode OUT, a template entry for that parameter does not appear, and upon successful completion of the procedure, a new variable would be residing in the PPA.

B.5 COMMAND PROCEDURE EXAMPLES

Several examples are provided in this section to show the capabilities of the command language, especially Command Procedures. In all examples, Command Language reserved words are indicated in lower case letters.

```
procedure PRINT_FILES
               (FILE_LIST:        in string;
                NUMBER_OF_COPIES: in integer := 1;
                ANSI_CONTROL :    in string := "Y";
                LISTING_DEVICE :  in string )           is

procedure PF2 (FILES:   in string :=  LASTFILE;
               COPIES:  in integer := 1;
               ANSI :   in string := "Y";
               DEVICE : in string := "LP02")
renames PRINT_FILES;

  LISTEMPTY : boolean := FALSE;
  CURRFILE : string;

begin
  loop

-- SPLIT the string of file names

    SYS.SPLIT(FILE_LIST,CURRFILE,LISTEMPTY);
    exit when LISTEMPTY;     -- exit when all files processed

-- Print each file as many times as required

    for I in 1..NUMBER_OF_COPIES loop
      SYS.PRINTCONTROL(CURRFILE,ANSI_CONTROL,LISTING_DEVICE);
    end loop;
  end loop;

-- show the output status for the output device requested

  SYS.PRINTSTAT(LISTING_DEVICE);
end PRINT_FILES;
```

Example B-3 COMMAND PROCEDURE TO PRINT FILES

In the previous example:

1. Command Interpreter primitives are indicated by the prefix "SYS.". Specific primitives are discussed in more detail in Section 12. The selected_component syntax of procedure_name is used for Primitives.

2. The semantics of the loop parameter in the FOR statement is identical to that defined in Ada.

3. "Short forms" of commands are possible through the use of *renaming declarations*. These renames may be contained within the command procedure itself or may occur in separate file(s). An unlimited number of renaming declarations may apply to a particular file. The semantics of the renaming declaration is identical to that defined in Ada. Different formal parameters and different default values may be defined, as illustrated in the preceding example. The procedure shown in Example B-3 could then be invoked by entering either:

```
PRINT_FILES( ...... );

    or

PF2( ...... );
```

A command procedure representing a possible method by which the Ada optimizing compiler could be interactively executed is:

```
procedure XADA (ANALYZER_CONTROL  : in string;
                OPTIMIZER_CONTROL,
                CODEGEN_CONTROL   : in string := "";
                OK                : out boolean)         is

begin

-- This procedure will handle the input of 1, 2 or 3 control
-- files. The user  may place all the compiler control
-- info in 1 file or break it up if desired. If 3 control
-- files are not specified, it is assumed that the first file
-- specified contains the required info.

  if OPTIMIZER_CONTROL = "" then
     OPTIMIZER_CONTROL := ANALYZER_CONTROL;
  end if;

  if CODEGEN_CONTROL   = "" then
     CODEGEN_CONTROL   := ANALYZER_CONTROL;
  end if;

  ANALYZE(ANALYZER_CONTROL);
  COMPILER_STATUS(ANALYZER_CONTROL,OK);

  if OK then
     OPTIMIZE(OPTIMIZER_CONTROL);
     COMPILER_STATUS(OPTIMIZER_CONTROL,OK);
  else
     return;       -- terminate compilation; exit procedure
  end if;

  if OK then
     CODEGEN(CODEGEN_CONTROL);
  end if;

end XADA;
```

Example B-4 COMMAND PROCEDURE for ADA COMPILATION

The compiler requires as input the path name of one control file. Optionally the user may supply three control files, one for each pass. A control file contains all the input parameters for a compiler pass, namely, the pathname of a library file, a list of compilation units to be processed, language pragmas, and pathnames of the output files. Each compiler phase is invoked via a procedure call. At the end of each phase, a compiler status program is invoked. It interrogates error files (possibly) generated by a pass and interactively reports the error status to the user. The database names of the error files associated with a compilation unit are obtained via the library file utility from the entry of the compilation unit in the library file. The status program prompts the user as to whether or not to continue the compilation. The user's decision is encoded in the variable, OK, and passed back to the Command Procedure. Continuation of the compilation is conditional on the value of OK. If it is false, execution of the procedure (and therefore the compilation) is terminated; otherwise, execution continues.

A command procedure representing a possible method by which a program may be compiled, bound, loaded and executed is shown next. This procedure requires the input of a source file name and optionally, a name to assign to the bound program file. All other activities, including control file generation, are transparent to the user. This example specifically shows one command procedure being called from another and also introduces the SYS.EXECUTE primitive.

```
procedure COMPILE_AND_BIND_AND_LOAD_AND_GO
          (SOURCE_NAME  : in string;
           PROGRAM_NAME : in string := "")          is

procedure CLG (SRC : in string;
               OBJ : in string := "")
renames   COMPILE_AND_BIND_AND_LOAD_AND_GO;

  COMPILER_CONTROL,
  PROGRAM_NAME,
  BINDER_CONTROL     : string;
  OK                 : boolean := FALSE;

begin

-- Build default control files for compiler and binder.
-- All compiler control info is contained in one file

   BUILD_CONTROLS(SOURCE_NAME,PROGRAM_NAME,COMPILER_CONTROL,
       BINDER_CONTROL,OK);

-- Call the Ada compiler command procedure.
-- Compiler informs user of the pathname of the library file.

   if OK then   -- Using named parameters
      XADA(ANALYZER_CONTROL => COMPILER_CONTROL,OK => OK);
   end if;

-- Call the binder program. This procedure assumes the first
-- subprogram found in the source file is the main program.
-- Binder produces a bound program file indicated by the
-- parameter PROGRAM_NAME. If no PROGRAM_NAME was supplied,
-- a name will be created.

   if OK then
      BIND(BINDER_CONTROL,PROGRAM_NAME,OK);
   end if;

-- Execute the program

   if OK then
      SYS.EXECUTE(PROGRAM_NAME);
   end if;

end COMPILE_AND_BIND_AND_LOAD_AND_GO;
```

Example B-5 COMPILATION AND EXECUTION COMMAND PROCEDURE

B.6 CLI and the PPA

B.6.1 PPA Concepts

The Program Parameter Area (PPA) is an unbounded storage abstraction which stores all variables used by the CLI. The PPA contains two string components for every entity; the name of the entity and its value. Referencing a variable which is not in the PPA returns the empty string.

B.6.2 Object Declarations

Variables which the user wants to affect the global PPA must be declared individually to the Command Interpreter as object declarations. An object declaration has the same effect as a statement when input directly to the CLI. For example:

```
SAVECC : INTEGER := 5;
```

establishes SAVECC in the global PPA, with an initial value of 5.

If no initialization accompanies the object declaration, no initial value is presumed.

B.6.3 Command Procedures

Local variables declared within the body of a command procedure are maintained in the PPA which accompanies the execution of that procedure. All variable references within a command procedure, whether they are local variables or default values for formal parameters shall reference the PPA. The PPA associated with execution of a command procedure or an Ada program is a copy of the PPA associated with the task that invoked it. If a command procedure contains a call to another procedure or program, a copy of the current PPA (which could contain local variables) is created and passed on. As an example, let us assume that command procedure PRINT_FILES, shown in Example B-3 was invoked by CLI directly, rather than being nested inside another Command Procedure. A copy of the PPA associated with the CLI is associated with the procedures to be executed. At the conclusion of the procedure, those variables which are designated as IN OUT or OUT parameters will be copied back into the parent's PPA (this is a form of implicit object declaration) and the current PPA will be deleted. Local variables are not copied back.

B.7 RULES OF LIBRARY SEARCHING

A basic system capability is the ability to create command libraries and also establish an explicit library search order. This search order can be changed at the user's discretion and may or may not include those functions provided as part of the Command Interpreter. When a procedure call is encountered, the currently defined search order is followed unless the procedure called is a primitive (indicated by the prefix SYS.), or the user specifically uses a procedure_name which has the syntax of a selected_component.

When a selected_component procedure name (qualified name) is used in a command procedure or via direct command entry, the command interpreter considers everything prior to the last period as the library to be searched, and the remainder is consider the file name. As such, a one time override of the user's library search order occurs. Additionally, any command(s) found within the invoked command file are interpreted within the context of the qualified name, unless the command is itself a qualified name, or a primitive. In other words, the qualified name is temporarily regarded as part of the library search list until the particular command is completed.

```
procedure PF ( .......) is
begin
    .
    .
  CLG ( ... );
    .
    .
  SYS.PRINTSTAT( ... );
  CHARLIE.SFG;
end PF;
```

Example B-6 LIBRARY SEARCH EXAMPLE

As an example, if the procedure shown in Example B-6 is in library TRYOUT.PROC and TRYOUT.PROC is not normally in the search list, this procedure would be invoked by the calling:

TRYOUT.PROC.PF (...);

and the following actions would occur:

1. The library TRYOUT.PROC is searched for PF. If it exists then it is invoked. If not, the user is notified of an error. No search of the normally defined libraries is performed.

2. When the call to CLG occurs, that call is accomplished in the context of the present search library. In other words,

TRYOUT.PROC is searched first, and if a version of CLG is found, it is invoked. If CLG is not found, then the normal search libraries are searched. If no version is found, execution of the procedure PF is halted.

3. After CLG is executed, SYS.PRINTSTAT is recognized and executed as a primitive.

4. When CHARLIE.SFG is encountered, since it is also a qualified name, the search WILL NOT be of the TRYOUT.PROC library. The library CHARLIE is searched, and if the file SFG contains any other procedure calls, they are resolved in the context of CHARLIE unless they themselves are qualified names.

B.8 COMMAND PROCEDURE INSTALLATION TOOL

All Command Procedures must be processed through the Command Procedure Installation Tool (CPIT) before they can be executed. Among the functions accomplished by the CPIT are:

1. Creation of a data structure analagous to the structure created by the Binder for Ada programs. This data structure is created from the command procedure specification and used when the procedure is called to accomplish type checking, parameter matching and possibly template generation. It contains the following information:

   a. parameter names

   b. parameter type information

   c. parameter mode indications

   d. parameter initial default values

2. Recognition of program attribute value assignments within the procedure. Program attributes are described in more detail in paragraph B.13. Recognition of program attributes results in the creation of a program attribute block which is used whenever the procedure is called.

3. Processing of RENAMES clauses. Procedure renaming declaration(s) can occur both within a command procedure and in separate files.

   a. When RENAMES are inside a command procedure, the CPIT will split the individual RENAMES into individual command procedure files. The name of each file is the same as that indicated in the RENAMES declaration, and is required because each command issued to the CLI results in a search for a specific file name. Each file created includes the appropriate parameter data structure (item 1 above) and a reference to the main procedure. This results in a separation of the procedure specification and the procedure body, and the possible association of many procedure specifications with a single procedure body. When one of the renamed commands is called into execution, the appropriate file is processed. It references the appropriate procedure body to execute.

b. When a renaming declaration(s) exists in a file by itself, the same process occurs. However, the user will be required to supply the full library name of the procedure which the renaming declaration references.

The processing described above will cause a transformation of the source form of the command procedure into an intermediate form. The forms are differentiated by means of a file attribute. The source form and the intermediate form(s) are related in the database. A change in the source form requires reprocessing by the CPIT.

B.9 TEMPLATE CREATION

A template may be constructed after CLI accesses a file containing either an Ada program or a command language procedure. This file contains a data structure created by either the Binder or Command Procedure Installation Tool, respectively. The CLI accesses this data structure as the first step in processing any command. The information contained in this structure is used to accomplish type checking, parameter matching and the construction of a template if required.

It should be noted that since the template is derived from the program file or command procedure, the use of meaningful procedure and parameter names in the procedure specification can be of great value. The template generator function displays underscores in variable names as blanks. In Example B-3, the parameter ANSI_CONTROL becomes ANSI CONTROL? on the template, and HOW_MANY_COPIES becomes HOW MANY COPIES?. Any initial (default) values are also displayed.

B.10 MESSAGE CAPABILITIES

Message capabilities are provided to transmit data to a user as:

1. Strings
2. Boolean values
3. Integer values

Messages are sent to the user by using the CLI primitive SYS.PUT. Messages are received from the user by using the CLI primitive SYS.GET.

B.11 PRIMITIVES

Primitives are functions which are accomplished directly by the Command Language Interpreter. Primitives are indicated by the prefix "SYS." in the procedure name. The following is a list of system primitives at this time:

SYS.SPLIT	splits a string containing commas into independent elements, so each can be accessed seperately. There are 4 parameters: The input string, a string into which a single list element is returned, a string in which the remaining list is retained (so the original list is not altered), and a flag which indicates when the list has been completely processed.
SYS.DATA	allows inclusion of data, which is not interpreted as commands, inside the body of a command procedure. There are two parameters: the name of a file where the data will be temporarily stored during execution of the command procedure, and an optional second parameter specifying a character which, when found at the beginning of a token, specifies that the token is to be expanded to reflect PPA values.
SYS.END_DATA	terminates the data stream started by SYS.DATA.
SYS.DATE	returns the data as MMDDYY. Requires one parameter of type STRING.
SYS.TIME	returns the time as HHMMSS on a 24 hour clock basis. Requires one parameter of type STRING.
SYS.JULIAN	returns the Julian date as YYDDD. Requires one parameter of type STRING.
SYS.PUT	outputs a variable, or message. One parameter is allowed, which can be any of the predefined types. This is the equivalent of PUTs defined in Ada.
SYS.GET	inputs a string. One parameter is allowed, which can be any of the predefined types. This is the equivalent of GETs defined in Ada.
SYS.EXECUTE	accepts one parameter of type STRING. This parameter is treated as a command and will be invoked by SYS.EXECUTE.

B.12 RUN Statement

The RUN statement provides an orderly means of bracketing a procedure call with the parameters needed for execution. RUN may be used for the following purposes:

1. Provides a way of executing a sequence of commands without creating a command procedure.
2. Provides a means of stating file and program attribute values applicable to the specific execution. This capability is detailed in paragraph B.13.

RUN statements may be dynamically nested. For example, if the following command were input,

```
RUN PF; END RUN;
```

PF could have RUN statements embedded in it. In this way, RUN functions similarly to a procedure call .

Since RUN is a compound statement, it may appear anywhere that a statement may appear. Thus sequences like those shown in Example B-7 are possible.

```
if <condition> then                   while I <= 5 loop
   run                                  run
     ....            and                  ...
   end run;                             end run;
end if;                               end loop;
```

Example B-7 RUN Statements

B.13 FILE AND PROGRAM ATTRIBUTES

The command language provides the capability to use file and program attributes in two different circumstances:

1. Within a command procedure
2. Within the RUN statement

The syntax of the assignment statement was chosen as the most simple and consistent mechanism for the representation of attribute assignments. It is used in either of the above cases.

B.13.1 Use in Command Procedures

Within a Command procedure it may be desirable to establish resource limits for processing or output. Typical program attributes are shown in Example B-8.

```
MAXTIME          -- cpu time allowed
MAXLINES         -- # of print lines allowed
```

Example B-8 TYPICAL PROGRAM ATTRIBUTES

File attributes can not be standardized, each one of them will require a reference to the Data Base Dictionary.

If a 60 second CPU time limit and a 2000 line print limit were desired on the PRINT_FILES command it could be handled as shown in Example B-9.

```
procedure PRINT_FILES
              (FILE_LIST:        in string :=  LASTFILE;
               NUMBER_OF_COPIES: in integer := 1;
               ANSI_CONTROL :    in string := "Y";
               LISTING_DEVICE :  in string := "LP02")
is

  LISTEMPTY : boolean := FALSE;
  CURRFILE : string;

begin

-- Program attribute assignment
  PRINT_FILES'MAXTIME := 60;
  PRINT_FILES'MAXLINES := 2000;

  loop

-- SPLIT the list of file names

    SYS.SPLIT(FILE_LIST,CURRFILE,LISTEMPTY);
    exit when LISTEMPTY;

-- Print the requested file as many times as required

    for I in 1..NUMBER_OF_COPIES loop
      SYS.PRINTCONTROL(CURRFILE,ANSI_CONTROL,LISTING_DEVICE);
    end loop;
  end loop;

-- show the output status for the output device requested

  SYS.PRINTSTAT(LISTING_DEVICE);
end PRINT_FILES;
```

Example B-9 COMMAND PROCEDURE with PROGRAM ATTRIBUTES

Attribute assignments are constrained as follows:

1. The procedure name must be present.
2. The attribute name specified must be valid.
3. Attribute assignment statements may appear anywhere in the procedure.
4. Use of program attributes results in the creation of a program

parameter block after the command procedure has been processed by the Command Procedure Installation Tool (CPIT).

5. Program parameters specified in a command procedure are overriden by the appearance of the same program parameters in a RUN statement that invokes the procedure.

B.13.2 Use in RUN Statement

Within the RUN statement, attribute assignments would be specified as shown in Example B-10.

```
RUN DOIT;                                RUN PRINT_FILES;
  ABC'TITLE := "XYZ";                      PRINT_FILES'MAXLINES := 3000;
  ABC'KIND := DISC;          (or)        END RUN;
  DOIT'MAXTIME := 300;
END RUN;

 (Ada program)                              (Command Procedure)
```

Example B-10 RUN COMMAND with ATTRIBUTES

In the preceding examples, both file and program attributes are utilized. The following points apply:

1. File attribute assignments have the same syntactic form as program attributes. They are distinguished as follows: Program attributes must have the same name (in the syntactic sense) as the procedure name being run. Otherwise, it will be regarded as a file attribute, placed in the File Parameter Block, and not referenced during execution.

2. DISC is not quoted in the previous example because it represents an enumerated value for the attribute KIND. While the Command Language does not support enumerated data types, DISC is enumerated for and by the Database Subsystem and will be recognized as such.

3. File attribute assignments cause the CLI to create a File Parameter Block (FPB). The FPB is merged at execution with the file parameter block created by the Ada compiler into a file attribute block.

4. Program parameters cause the CLI to create a program parameter block which is merged at execution with the program parameter block created by the CPIT into a program attribute block.

5. File and Program attribute assignments can be intermixed; there is no order requirement.

6. If the attribute does not exist (not a predefined program attribute and not a database attribute as listed in the dictionary), the user is notified of the error and the RUN statement is NOT executed.

7. Program and file attribute assignments are meaningful only within the context of a RUN statement and a Command Procedure.

B.14 OVERVIEW: COMMAND LANGUAGE SYNTAX

The same syntactical notation used in the Ada language definition is also used in this section, with the exception that reserved words are capitalized.

The grammar used is LL(1), except as indicated in the following cases.

1. Choice: A choice can be a simple expression, a range, or the keyword OTHERS. A range is defined as simple_expression .. simple_expression. The difference between a simple_expression and a range can not be determined until the presence of the .. is ascertained.

2. Command: A Command can be either a statement, command_procedure or a declaration. Two problems exist here:

   a. A statement and declaration can both begin with an identifier. This situation is fairly complicated because the difference between a procedure_call, assignment_statement, loop_identifier and declaration must be determined. Differentiation involves looking for the "=" part of the := , finding the type mark, the "(", or the keywords LOOP, WHILE, or FOR.

   b. A command_procedure and declaration can both begin with a command_procedure_specification. This is because a declaration can be a RENAMING declaration. The difference can not be determined until the token IS or RENAMES is found.

3. Name: A name can be an identifier, attribute, or selected_component. Attributes and selected_components are both defined in terms of name. Two problems exist here:

   a. An identifier can not be differentiated from a selected_component or attribute until the "." or ' is found.

b. The difference between an attribute and a selected_component can not be ascertained until the ' mark is found.

4. Parameter Association: It is impossible to tell whether one has a formal_parameter or actual_parameter until the presence of the "=>" is ascertained. This conflict is handled by looking ahead, or by exploiting the slight difference between the 2 entities. A formal_parameter is an identifier only, while an actual_parameter is an expression.

5. Simple_statement: A simple_statement can be either a null_statement, assignment_statement, return_statement, procedure_call, or exit_statement. An assignment_statement and procedure_call both begin with a name, so it is not possible to tell the difference until either the (or := is found.

6. Statements: A statement can be either a simple_statement or a compound_statement. Both can begin with an identifier, as occurs with LOOP statements and assignment statements. The difference can not be determined until the "=" part of the ":=" is found.

The following syntax will be determined and controlled by the lexical analyzer and as such, the following symbols will be considered as terminal symbols in the command language grammar.

```
identifier ::= letter {[underscore] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

integer ::= digit {[underscore] digit}
```

B.15 COMMAND LANGUAGE SYNTAX

```
character_string ::= "{character}"

declaration ::= object_declaration |
                renaming_declaration

object_declaration ::= identifier_list :  type_mark
                       [:= expression  ]

identifier_list ::= identifier {,identifier}

type_mark ::= INTEGER   |
              BOOLEAN   |
              STRING

name ::= identifier | attribute | selected_component

   Note: This allows simple variables (in the PPA),
         attributes and qualified procedure names

attribute ::= name'identifier

   Note: In combination with name, this allows for file attributes,
         program attributes, and possibly variable attributes

selected_component ::= name.identifier

literal ::= integer |
            character_string |
            NULL

expression ::= relation { logical_operator relation }

relation ::= simple_expression
             [relational_operator simple_expression]

simple_expression ::= {unary_operator} term {adding_operator term}
```

```
term ::= factor {multiplying_operator  factor}

factor ::= primary [ ** primary ]

primary ::= literal |
            name |
            (expression)

logical_operator ::= AND | OR | XOR | AND THEN | OR ELSE

  Note: As in Ada, the logical operator used can not be
        varied unless the expression is parenthesized

relational_operator  ::=   = | /= | <  | <= | > | >=

adding_operator ::= + | - | &

unary_operator ::=  + | - | NOT

multiplying_operator ::=  * | / | MOD | REM

exponentiating_operator ::= **

choice ::= simple_expression | range | OTHERS

range ::= simple_expression..simple_expression

renaming_declaration ::=  command_procedure_specification
                          RENAMES name;

command_procedure_specification ::= PROCEDURE identifier
                                    [formal_part]

command ::=  statement  |
             command_procedure  | declaration

statement  ::= simple_statement | compound_statement

sequence_of_statements ::= statement {statement}

simple_statement ::=   null_statement  |
                       assignment_statement  |
                       return_statement  |
                       procedure_call  |
                       exit_statement
```

```
compound_statement ::= if_statement |
                       loop_statement |
                       case_statement |
                       run_statement  |
                       BEGIN sequence_of_statements END;

null_statement ::= NULL;

assignment_statement ::= variable_name := expression;

   Note: variable_name conforms to the same syntax as name

return_statement ::= RETURN;

   Note: Since functions are not allowed, the optional expression
         part of the RETURN statement is omitted

if_statement ::=  IF condition THEN
                     sequence_of_statements
                  {ELSIF condition THEN
                     sequence_of_statements}
                  {ELSE
                     sequence_of_statements}
                  END IF;

condition ::= boolean_expression

   Note: boolean_expression conforms to the same syntax as expression

case_statement ::= CASE expression IS
                     {WHEN
                        choice {| choice} => sequence_of_statements}
                   END CASE;

loop_statement ::=  [loop_identifier:]
                    [iteration_clause]
                    LOOP
                      sequence_of_statements
                    END LOOP
                    [loop_identifier];

  Note: loop_identifier conforms to the same syntax as identifier
```

```
iteration_clause ::= FOR loop_parameter
                     IN [REVERSE] range  |
                     WHILE condition

loop_parameter ::= identifier

exit_statement ::= EXIT [loop_identifier] [WHEN condition];

run_statement ::= RUN  procedure_call  {attribute_statement}
                  END RUN;

attribute_statement ::=  attribute := expression;

command_procedure ::= command_procedure_specification IS
                        {declaration}
                      BEGIN
                        sequence_of_statements
                      END [identifier];

formal_part ::= (parameter_declaration {; parameter_declaration})

parameter_declaration ::= identifier_list : mode type_mark
                          [:= expression]

mode ::= [IN] | OUT | IN OUT

procedure_call ::= procedure_name [actual_parameter_part];

   NOTE: procedure_name conforms to the same syntax as name
         and can be either a command procedure, program or CLI
         primitive

actual_parameter_part ::= (parameter_association
                          {, parameter_association})

parameter_association ::= [formal_parameter =>] actual_parameter

formal_parameter ::= identifier

actual_parameter ::= expression

  Note: Comments have the same syntax as in Ada
```

ATTACHMENT 1 -- REFERENCES

[BRE80] Brender, R., The Case Against Ada as a APSE Command Language, SIGPLAN Notices, (Oct 80), 27-32.

[BUR77] B 7000/B 6000 I/O Subsystem Reference Manual, Burroughs Corporation, (1977).

[BUR78A] B 7000/B 6000 CANDE Reference Manual, Burroughs Corporation, 1978.

[BUR78B] B 7000/B 6000 Work Flow Language Reference Manual, Burroughs Corporation, 1978.

[JOHN] Johnson, Kolberg, and Sinnamon, A Programmable System for Software Configuration Management, Texas Instruments.

[MoD] United Kingdom Ministry of Defense, Ada Support System Phase 2 and 3 Reports.

[SCH80] Schofield, Hillman, and Rodgers, MM/1, A Man-Machine Interface, Software-Practice and Experience, 10, 751-763.

[TI79A] Model 990 Computer DX10 Operating System Release 3.3 Reference Manual, Volume 5 System Programming Guide, 2-1 through 2-40, Texas Instruments Inc, (December 1979).

[TI79B] Model 990 Computer DX10 Operating System Release 3.3 System Design Document, Texas Instruments Inc, (December 1979).

[TUT80] Tutleman, J.S., Handling Large, Complex and Variable Job-Control Streams using a Procedure Invocation Package (PIP) and a Pseudo Library, SIGPLAN Notices, (Dec 80), 85-91.

[UNIX78A] Ritchie, D. M. and Thompson, K., The Unix Time Sharing System, Bell System Technical Journal, Vol 57, No 6, (Jul-Aug 1978), 1905-1929.

[UNIX78B] Thompson, K., Unix Implementation, Bell System Technical Journal, Vol 57, No 6, (Jul-Aug 1978), 1931-1946.

[UNIX78C] Ritchie, D. M., A Retrospective, Bell System Technical Journal, Vol 57, No 6, (Jul-Aug 1978), 1947-1969.

[UNIX78D] Bourne, S. R., The Unix Shell, Bell System Technical Journal, Vol 57, No 6, (Jul-Aug 1978), 1971-1990.

[WARD] Ward, D. and Ward, R., Features of General Purpose Timesharing Systems.

APPENDIX C

INPUT / OUTPUT INTERFACE

The high-level I/O interface supplied in KAPSE is based on the packages INPUT_OUTPUT and TEXT_IO presented in section 14 of the Ada reference manual [DOD80B]. This appendix describes how those packages are modified.

C.1 Package INPUT_OUTPUT

In the original version of INPUT_OUTPUT, the first operation that is applied to a file is CREATE or OPEN, each of which initializes file data structures and uses a string parameter to connect to an external file; attributes of a file are encoded within its (generalized) name. The modified version separates these functions so file attributes can be specified individually prior to connection. With this approach a new attribute can be supported incrementally by adding subprograms to set and query the attribute.

```
procedure INITIALIZE( FILE : in out IN_FILE );
procedure INITIALIZE( FILE : in out OUT_FILE );
procedure INITIALIZE( FILE : in out INOUT_FILE );
```

This subprogram initializes the internal data structures associated with FILE in preparation for a call to CREATE or OPEN; the state of FILE is closed.

```
procedure DEALLOCATE( FILE : in out IN_FILE );
procedure DEALLOCATE( FILE : in out OUT_FILE );
procedure DEALLOCATE( FILE : in out INOUT_FILE );
```

This subprogram must be called to deallocate the internal data structures that are associated with FILE. The exception STATUS_ERROR is raised if FILE is not closed.

```
procedure SET_NAME( FILE : in IN_FILE;
                    NAME : in STRING );
procedure SET_NAME( FILE : in OUT_FILE;
                    NAME : in STRING );
procedure SET_NAME( FILE : in INOUT_FILE;
                    NAME : in STRING );
```

The external name attribute of the internal file represented by FILE is set to the string NAME. The exception STATUS_ERROR is raised if FILE has not been initialized.

```
procedure CREATE( FILE : in OUT_FILE );
procedure CREATE( FILE : in INOUT_FILE );
```

An external file is created and associated with FILE, which becomes open. If the external name attribute of FILE has not been set, then a (unique) temporary file is generated. The exception NAME_ERROR is raised if an external file with the specified name already exists, the user does not have appropriate access rights, or the file name has improperly specified. The exception STATUS_ERROR is raised if FILE has not been initialized or FILE is already open.

```
procedure OPEN( FILE : in IN_FILE );
procedure OPEN( FILE : in OUT_FILE );
procedure OPEN( FILE : in INOUT_FILE );
```

A connection is established between an external file and an internal file represented by FILE; the internal file becomes open. The exception NAME_ERROR is raised if an external file with the specified name does not exist or the user does not have appropriate access rights. The exception STATUS_ERROR is raised if FILE has not been initialized, FILE is already open, or the user does not have appropriate access rights.

```
procedure CLOSE( FILE   : in IN_FILE;
                 DELETE : in BOOLEAN := FALSE );
procedure CLOSE( FILE   : in OUT_FILE;
                 DELETE : in BOOLEAN := FALSE );
procedure CLOSE( FILE   : in INOUT_FILE;
                 DELETE : in BOOLEAN := FALSE );
```

The file FILE is disconnected from its associated external file and becomes closed. The exception STATUS_ERROR is raised if FILE has not been initialized or is already closed. NAME_ERROR is raised if the user does not have appropriate access rights to delete the external file.

Additional subprograms will be defined that set and examine file attributes. Since internal file structures are not deallocated by CLOSE, an attribute of an internal file is not lost if the file is closed and the same file variable is reused for another file.

The following example copies characters from teletypewriter TTY to printer PRT.

```
declare
  package CHAR_IO is
    new INPUT_OUTPUT( CHARACTER );
  use CHAR_IO;
  CH     : CHARACTER;
  INPUT  : IN_FILE;
  OUTPUT : OUT_FILE;
begin
  INITIALIZE( INPUT, "TTY" );
  OPEN( INPUT );
  INITIALIZE( OUTPUT, "PRT" );
  OPEN( OUTPUT );
  while not END_OF_FILE( INPUT ) loop
    READ( INPUT, CH );
    WRITE( OUTPUT, CH );
    end loop;
  CLOSE( INPUT );
  DEALLOCATE( INPUT );
  CLOSE( OUTPUT );
  DEALLOCATE( OUTPUT );
end;
```

C.2 Package TEXT_IO

The package TEXT_IO is extended to use pictorial formats to specify of conversion between the internal and external data representations. This capability, which is similar to features provided in PL/1 and COBOL, permits the specification for a collection of items to be given in one place and in a form that is easy to visualize.

A conversion format is specified by a string composed from the following characters (and blanks):

* A -- Alphabetic character

* N -- Alpha-numeric character

* S -- Leading sign

* Z -- Zero-suppressed number
* 2 -- Binary number
* 8 -- Octal number
* 9 -- Decimal number
* # -- Hexadecimal number
* $ -- Leading dollar sign
* . -- Location of decimal (radix) point
* X -- Exponent of real decimal

For example, the string "ZZZZZZ9.99 NNNN" could be used to print a zero-suppressed fixed point decimal number followed by a four-character alphanumeric.

The private type FORMAT is added to TEXT_IO to serve as the abstraction for a pictorial format. A conversion specification is changed from a string to a FORMAT by

```
procedure CREATE_FORMAT( PICTURE : in STRING;
                         NAME    : out FORMAT ).
```

This procedure checks the syntax of the specification and compiles it into an internal representation that can be processed more efficiently than the original string form. Each format is viewed as a series of conversion specifications and has an implicit position that identifies the next specification to be used. The format can be positioned to its first specification by

```
procedure RESET_FORMAT( NAME : FORMAT ).
```

When a format is no longer needed, it must be deallocated with

```
procedure DESTROY_FORMAT( NAME : in out STRING ).
```

For each data type that is supported by TEXT_IO, overloaded procedures are added that have a format as parameter; e.g.,

```
procedure GET (FILE     : in IN_FILE;
               PICTURE  : in FORMAT;
               VARIABLE : out INTEGER );

procedure PUT (FILE     : in OUT_FILE;
               PICTURE  : in FORMAT;
               VARIABLE : in INTEGER );
```

The following statements show the creation of a format associated with an employee data record and the use of that format to print information on the standard output file.

```
CREATE_FORMAT( "999999  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  $$$$$$9.99",
               EMPLOYEE_DATA );
--
RESET( EMPLOYEE_DATA );
PUT( EMPLOYEE_DATA, EMPLOYEE.NUMBER );
PUT( EMPLOYEE_DATA, EMPLOYEE.NAME );
PUT( EMPLOYEE_DATA, EMPLOYEE.SALARY );
--
DESTROY_FORMAT( EMPLOYEE_DATA );
```

APPENDIX D

TERMINAL INPUT / OUTPUT

This appendix describes the high-level interactive terminal I/O interface supplied by the ASE. High-level interactive terminal support is provided for teletypewriters and alphanumeric video display terminals (VDTs).

Several levels of support will be provided for these terminals; the applicability of a particular support package to a particular terminal depends on the capabilities of the terminal:

* Logical record I/O support is provided for all interactive terminals.
* Field level I/O support is provided for VDTs that are capable of logical record support and have an addressable cursor position.
* Screen Level I/O support is provided for
  - VDTs which communicate with host systems at the block level, such as an IBM 3270, and
  - VDTs for which field level I/O support is provided.

D.1 Logical Record I/O

Logical record support is provided for all interactive terminals and is the only level of support provided for teletypewriters. Logical record support is in many ways similar to the support provided for a text file. The terminal may be opened by a user as an IN_FILE, OUT_FILE, or INOUT_FILE.

When any input command is specified by the user, a logical record is obtained from the terminal. The logical record is terminated when the terminal user presses the return key. This logical record is treated by the user program as a line in a text file. For terminals that do not provide character/line deletion, control keys will be defined that perform these operations.

With the exception of CREATE, DELETE, and operations which refer to or cause a modification a modification to the position of an item within a file, all of the operations in the packages INPUT_OUTPUT and TEXT_IO may be performed on terminals with logical record support.

D.2 Field Level I/O

Field-level I/O support is provided for VDTs which are capable of transmitting/receiving logical records at a specified screen position, such as an ADM 3A or DEC VT100. Programs using field-level I/O transmit/receive strings of a specified length.

For both read operations and write operations, a program may specify the following attributes (if available on the terminal being used):

* the position on the screen of the first character of the field to be read/written,
* the position of the cursor within the field to be read,
* the color of the field,
* reverse video,
* brightness level, and
* blinking field.

In addition, on read operations a program may:

* specify which control characters (including the return key) may cause termination of the read operation,
* specify that the read operation should terminate when the field is filled,
* obtain the identity of the control key that terminated the read operation (if any), and
* obtain the position of the cursor when the read terminated.

D.3 Screen Level I/O

For terminals such as the IBM 3270 for which screen-level I/O support are provided, communication with their host system is by means of a screen format description. In other words, the program must specify the format of the screen and transmit that information to the terminal. The terminal then handles all screen modifications until a control key is pressed by the user. The terminal then sends information concerning the modifications made to the screen back to the host system.

The following information must be specified by a program using this level of terminal support.

* field positions,

* field lengths, and
* field protection.

Depending on the the capabilities of the particular terminal being used, a program may also specify the following attributes for a field:

* Color,
* Reverse video,
* Programmable brightness,
* Character and/or field blinking

A program may specify that the VDT screen be cleared and may obtain identification information indicating which control key terminated the read.

APPENDIX E

EXAMPLE TERMINAL SESSION

This appendix describes a session at a terminal in which a user logs on, compiles a program unit, (concurrently) edits another unit, and logs out. A series of figures depicts the screen of the user's terminal and the programs and tasks that are active in his environment.

E.1 User Enters APSE

A user typicallly enters APSE by performing the login sequence required by his particular host operating system and executing a host system command that executes the APSE executive program with connection to the user's terminal. From this point, the user communicates with APSE through the command language of APSE, not that of the host operating system.

Figure E-1 shows the state of the user's environment when the APSE executive program is activated by the host operating system. Seven tasks are active in the executive program. The user's terminal is connected to a version of the device controller that is appropriate for the particular type of terminal being used. The device controller is connected to the terminal controller, which by default comes up with two windows. A one-line window is connected to the log manager through its virtual terminal so system messages can be displayed as they are generated. The remainder of the physical screen is configured as a window in which the executive control task is active. A prompt of "!" to indicates the executive is in control and waiting for a command. The PPA manager and KAPSE interface tasks are idle.

E.2 User Logs In

E-2 shows the user's terminal at the point an APSE "login" command is entered for user TCH212. (The screen is partitioned into two windows whose headings indicate that the executive program is waiting for terminal input and the log manager is running (but waiting for an inter-program message).) The executive responds by prompting for a password and sending a message via the KAPSE interface task (KIT) to the APSE manager to verify the user's access rights. The APSE manager returns a copy of the user's permanent environment, which becomes the root of the PPA tree that is manipulated by the PPA manager. The state after login is shown in Figure E-3.

E.3 CLI Activated

Before the user can execute the Ada optimizing compiler, an instance of the command language interpreter (CLI) must be instantiated. This is done by responding to the executive prompt "!" with "?", the prompt used by CLI. This causes the executive to disconnect from its window and instantiate a copy of CLI with connection to the window that the executive vacated. This is shown in Figure E-4, in which CLI 1 has been created with a virtual terminal.

E.4 Ada Optimizing Compiler Activated

The user requests that the compiler be invoked by entering "compile", the name of the command language procedure for the compiler. The parameters of this procedure are the name of a library file and a list of units within it that are to be compiled. E-5 shows the user's screen with a request to compile units GET, GET_LINE, and GET_STRING of library file TEXT_IO. (This screen also illustrates that a CLI command may be entered on the same line as the "?" that switches from the executive to a new instantiation of CLI.) CLI responds to this command by invoking the first step of the compilation, the Ada analyzer. Figure E-6 shows the first step in the loading of this program. The canonical Ada program is loaded by CLI using a host operating system service request. This program begins execution with an instance of KIT in communication with CLI to receive messages to load the analyzer and initialize its program arguments. The virtual terminal is disconnected from CLI and given to the analyzer. After loading is complete, the analyzer begins execution (Figure E-7) and sends a message to its virtual terminal (E-8).

E.5 Editor Activated

Suppose the user decides to edit another program unit while the compilation is in progress. The virtual terminal of the compiler must be disconnected from its window so the compilation can proceed in the "background." (It would also be possible to split the compiler's window and display output concurrently from both programs.) The first step in activating the editor is to depress the break key, which causes the window (in which the cursor is positioned) to be disconnected from the virtual terminal of the compiler and connected to the executive control task, as is shown in Figure E-9. (The compiler is not halted by the "break"; it continues to execute until it blocks on an I/O request to its virtual terminal.) If the user responds to the prompt of the executive with '?edit("INPUT_OUTPUT.SOURCE")' (E-10), a second copy of CLI and its virtual terminal are instantiated and connected to the window (Figure E-11), and CLI uses the editor command procedure to load the editor and give it connection to the virtual terminal (Figure E-12 and Figure E-13). (At this point the, the tree of PPAs has five entries: a copy of the permanent PPA, one for each of the command procedures, and one for each of the active programs.) The terminal (E-14) displays the text for file "INPUT_OUTPUT.SOURCE" and a prompt for an editor command.

E.6 Compiler Terminates

When the analyzer terminates, the first copy of CLI is reactivated to continue interpretation of the compiler command procedure. When the remaining steps of the compilation process are complete, the command procedure terminates after sending a message to the log manager indicating the success of the compilation (E-15). This message is displayed automatically on the one-line window of the log virtual terminal so it can be observed by the user as he edits. The first CLI is suspended while waiting for the user to enter further commands at its virtual terminal (Figure E-16).

To return to the first copy of CLI, the user depresses "break" to disconnect the editor from the screen and enter the executive (Figure E-17). The executive command "attach" is entered (E-18) to reconnect the window to the virtual terminal of the first CLI (Figure E-19). At this point, the user could enter a command to invoke the binder to contruct a program to be tested. If this is not desired, a "quit" command (E-20) can be entered to terminate the first copy of CLI and cause the executive to be reconnected (Figure E-21). The editor is resumed entering "attach" (E-22) to request the executive to connect the window to the editor (Figure E-23 and E-24).

E.7 Editor Terminates

When the user finishes his edit session, he enters a "quit" command (E-25), which causes the editor to terminate and the virtual terminal to be returned to the second copy of CLI (Figure E-26).

E.8 User Logs Out

When CLI becomes active, it prompts for a command; if "quit" is entered, CLI terminates, and the executive control task becomes active and prompts with "!" (Figure E-27). If "logout" is entered (E-28), the root PPA is written back to the database so it can be restored the next time the user logs in. The APSE executive program enters the same state that it had when the user logged in to the host operating system (Figure E-29). At this point, another user can log in to APSE or a host system "logout" command can be issued to disconnect the terminal.

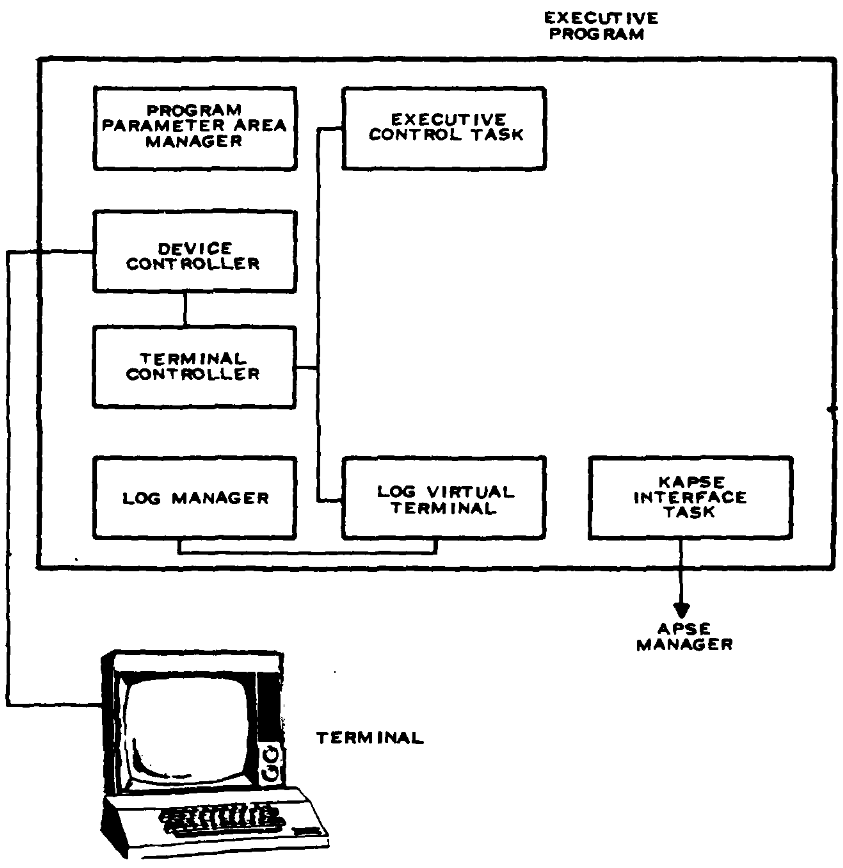
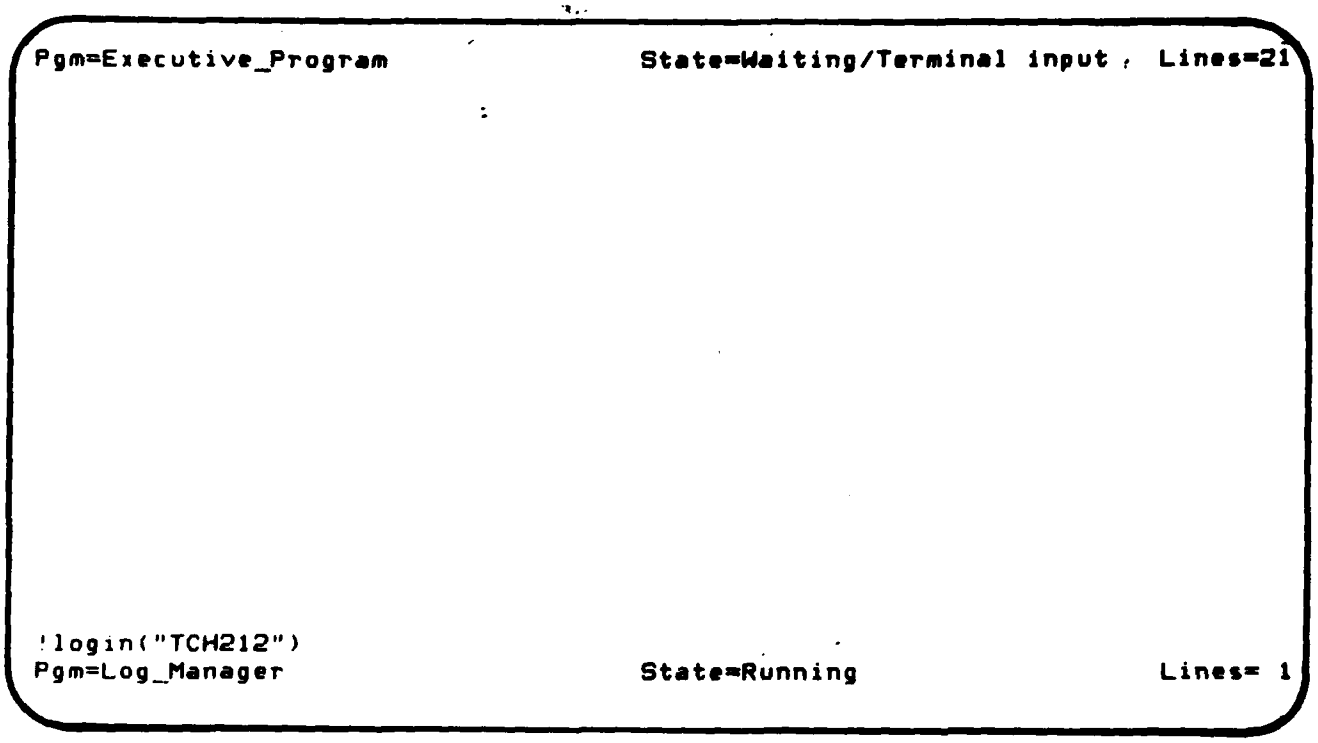


Figure E-1 User Enters APSE



```
Pgm=Executive_Program                    State=Waiting/Terminal input   Lines=21
















!login("TCH212")
Pgm=Log_Manager                          State=Running                  Lines= 1
```

Figure E-2 User Enters LOGIN Command

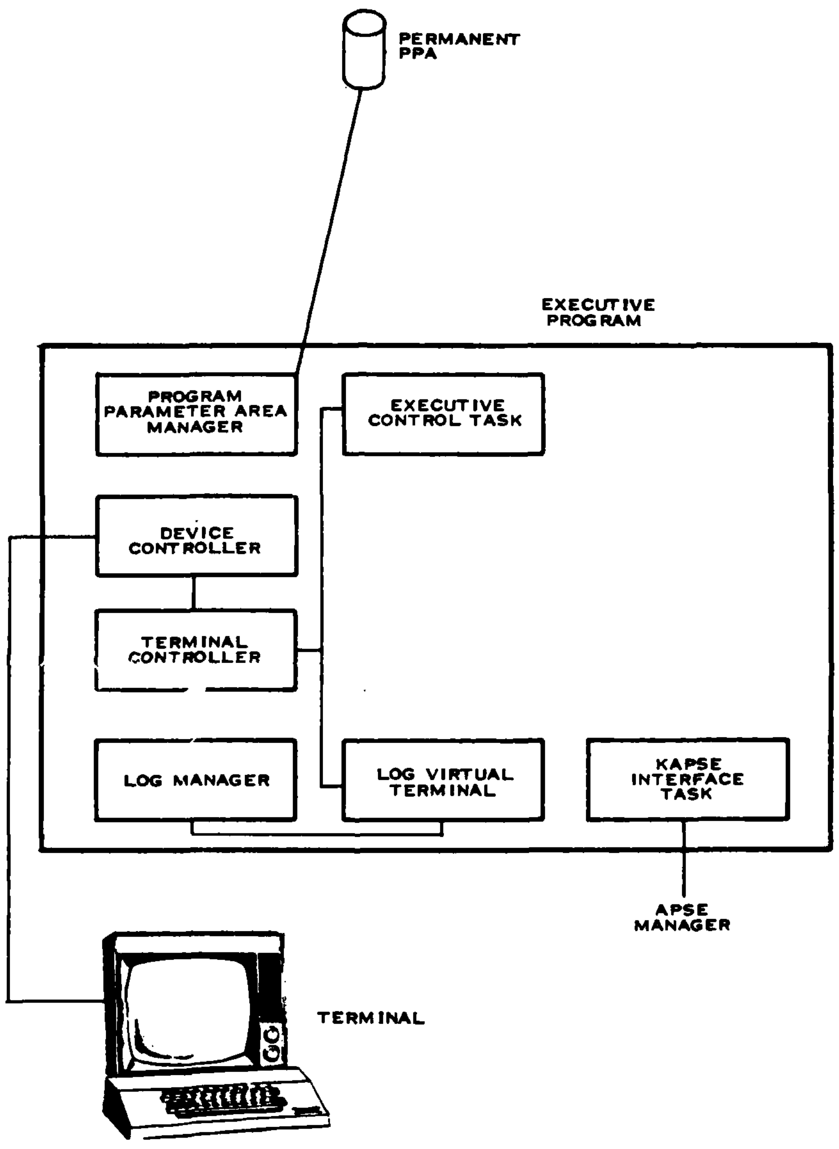


Figure E-3 Login Complete

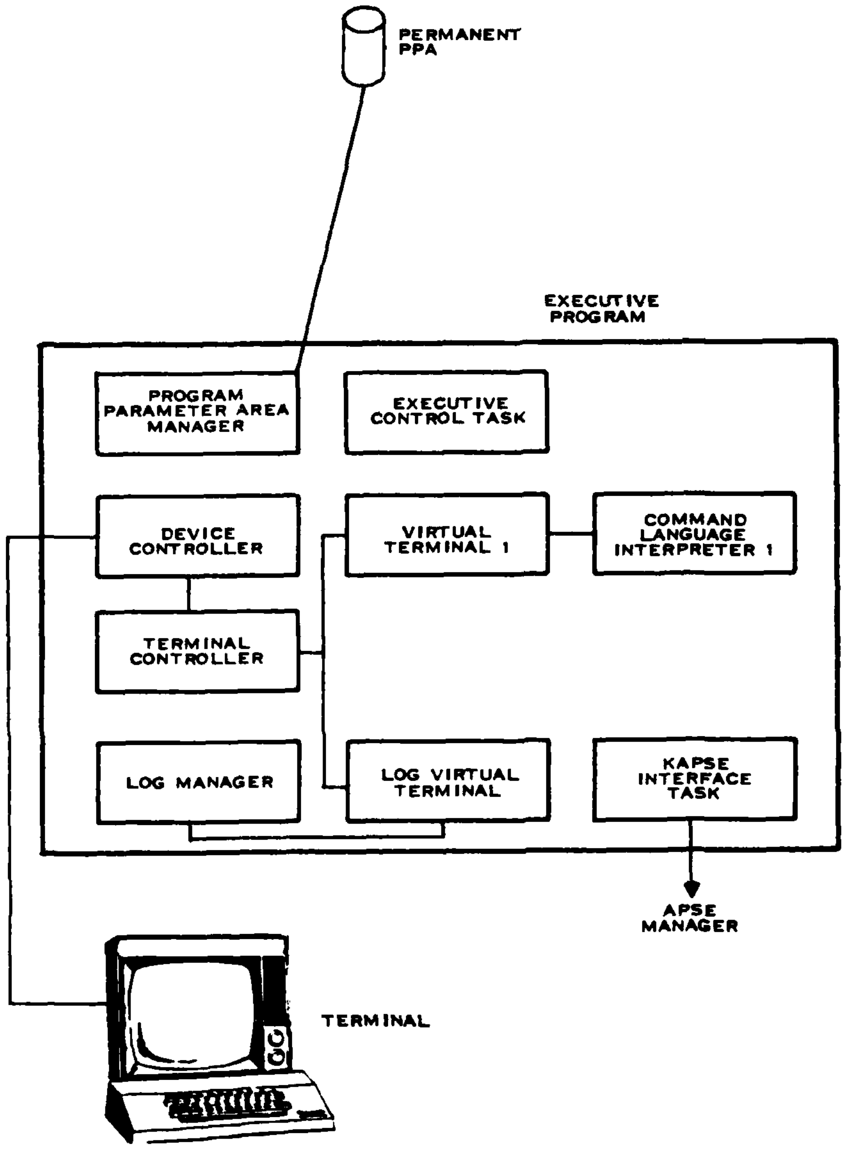
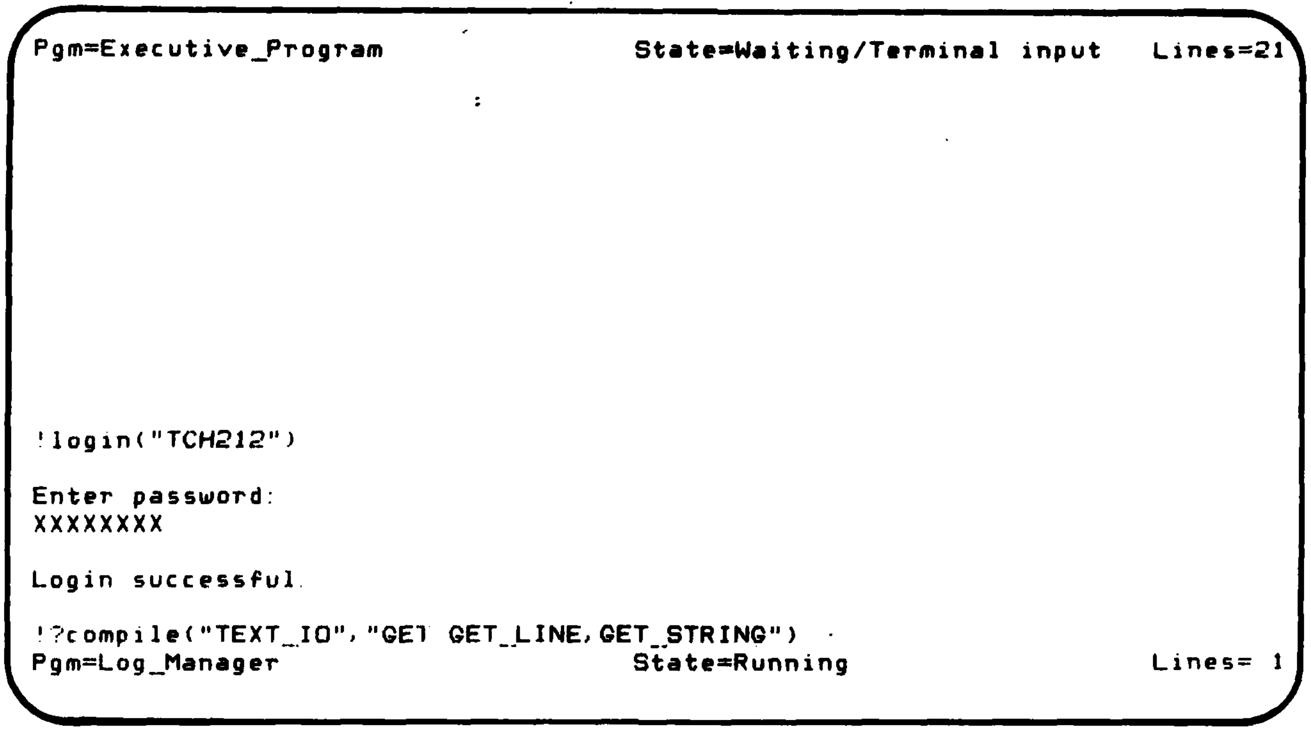


Figure E-4 CLI Activated



```
Pgm=Executive_Program               State=Waiting/Terminal input   Lines=21

!login("TCH212")

Enter password:
XXXXXXXX

Login successful.

!?compile("TEXT_IO","GET GET_LINE,GET_STRING")
Pgm=Log_Manager                     State=Running                  Lines= 1
```

Figure E-5 User Activates Compiler

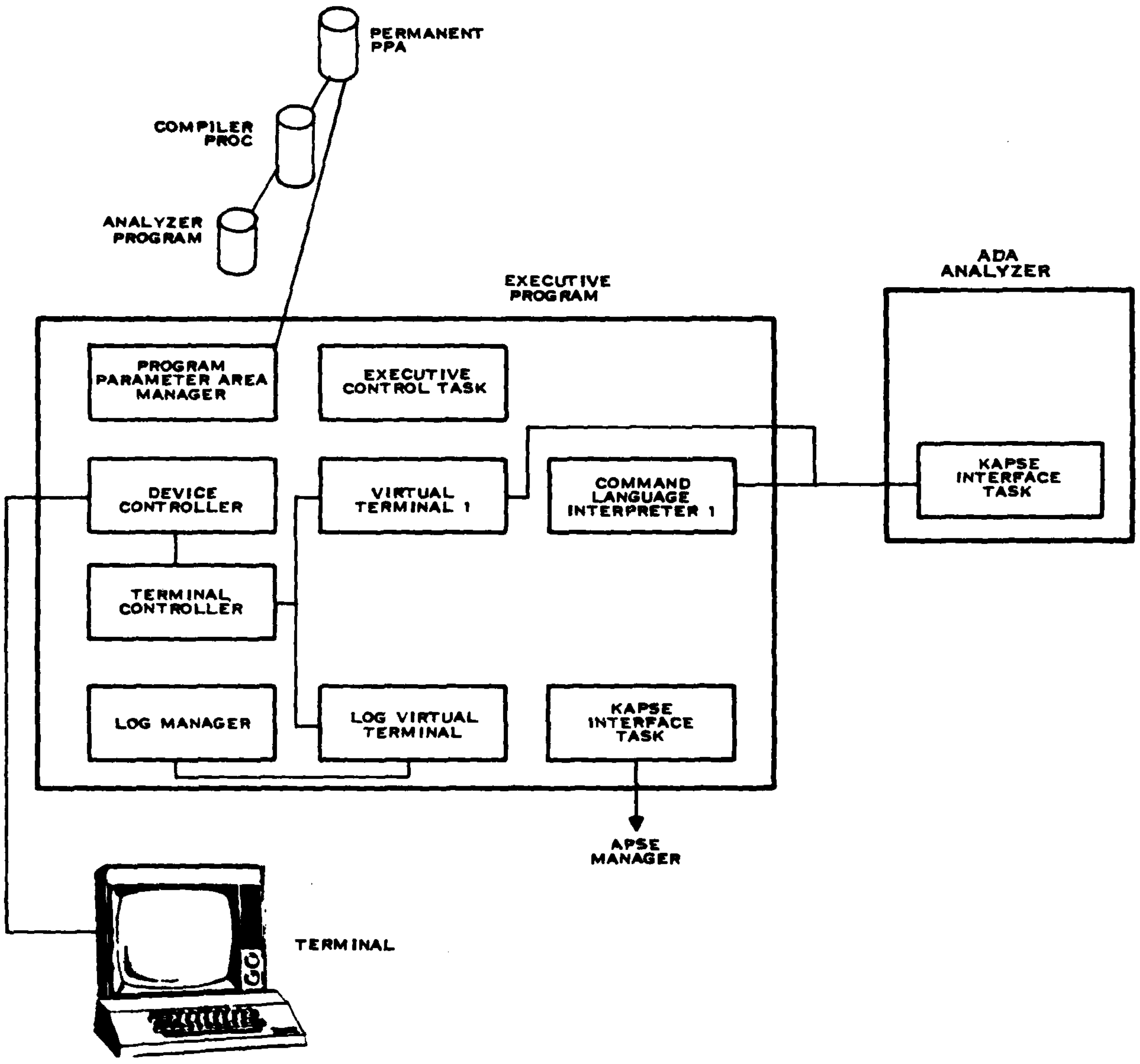


Figure E-6 Loading of Compiler Begins

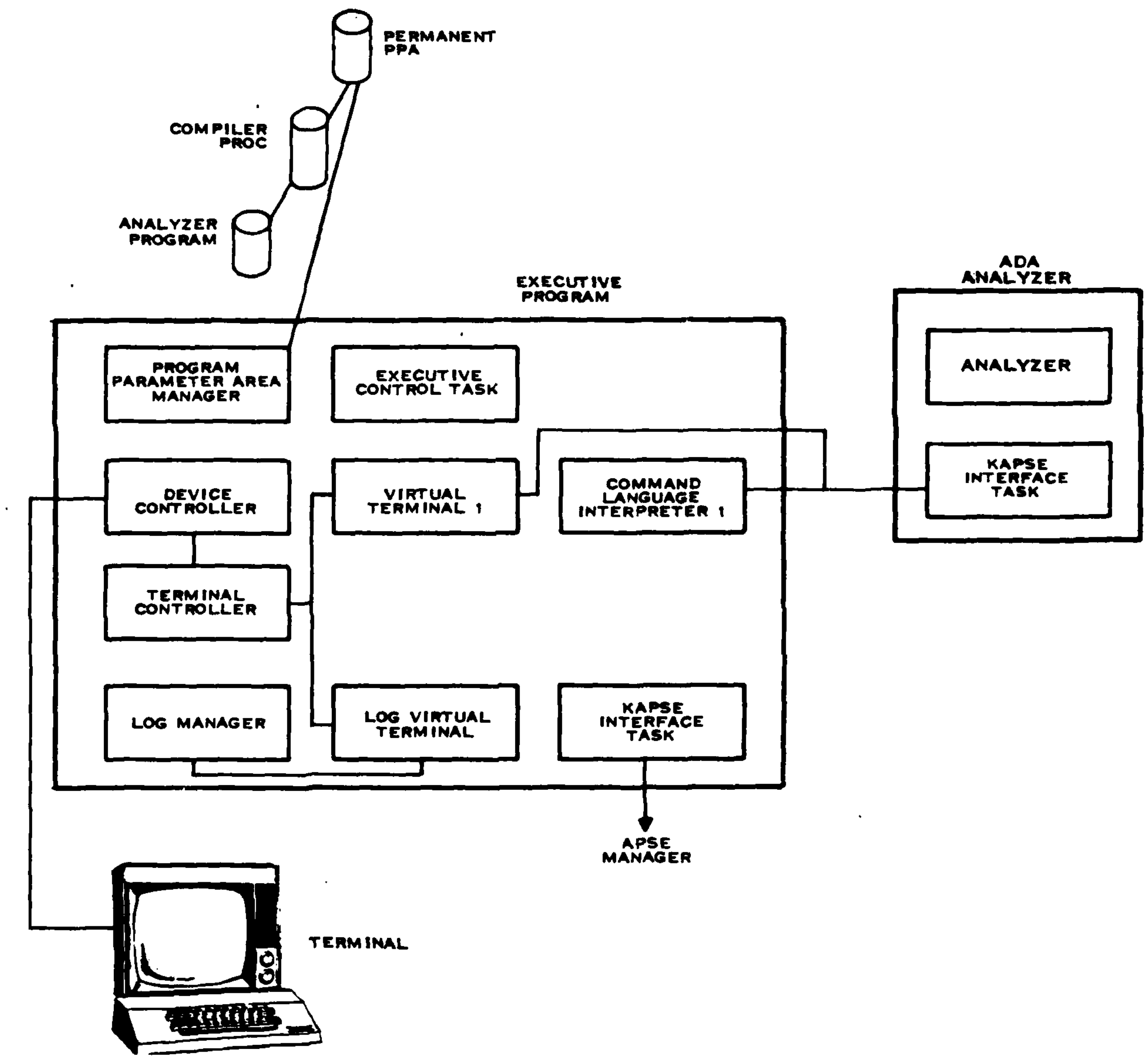
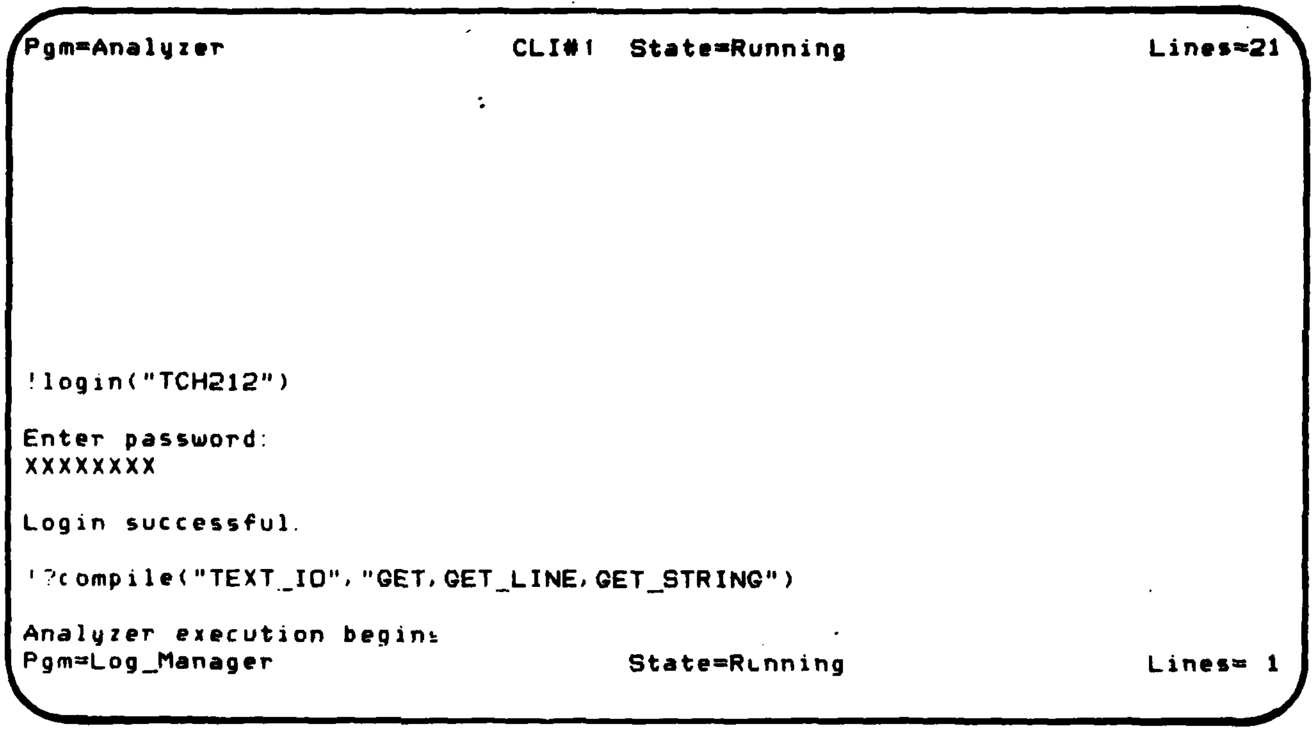


Figure E-7 Analyzer in Execution



```
Pgm=Analyzer                    CLI#1  State=Running                    Lines=21









!login("TCH212")

Enter password:
XXXXXXXX

Login successful.

!?compile("TEXT_IO","GET,GET_LINE,GET_STRING")

Analyzer execution begins
Pgm=Log_Manager                        State=Running                    Lines= 1
```

Figure E-8 Analyzer Prints First Output

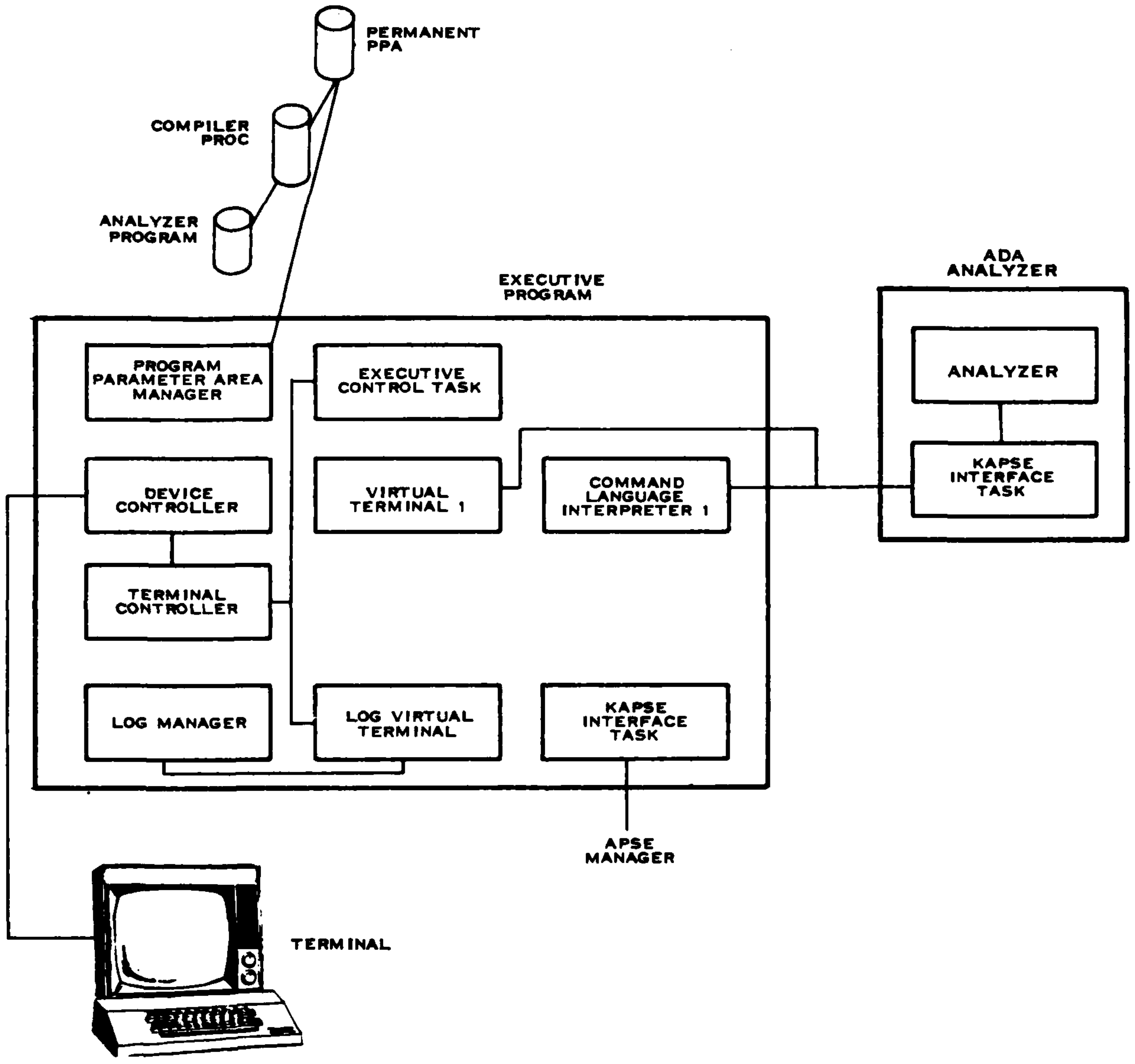
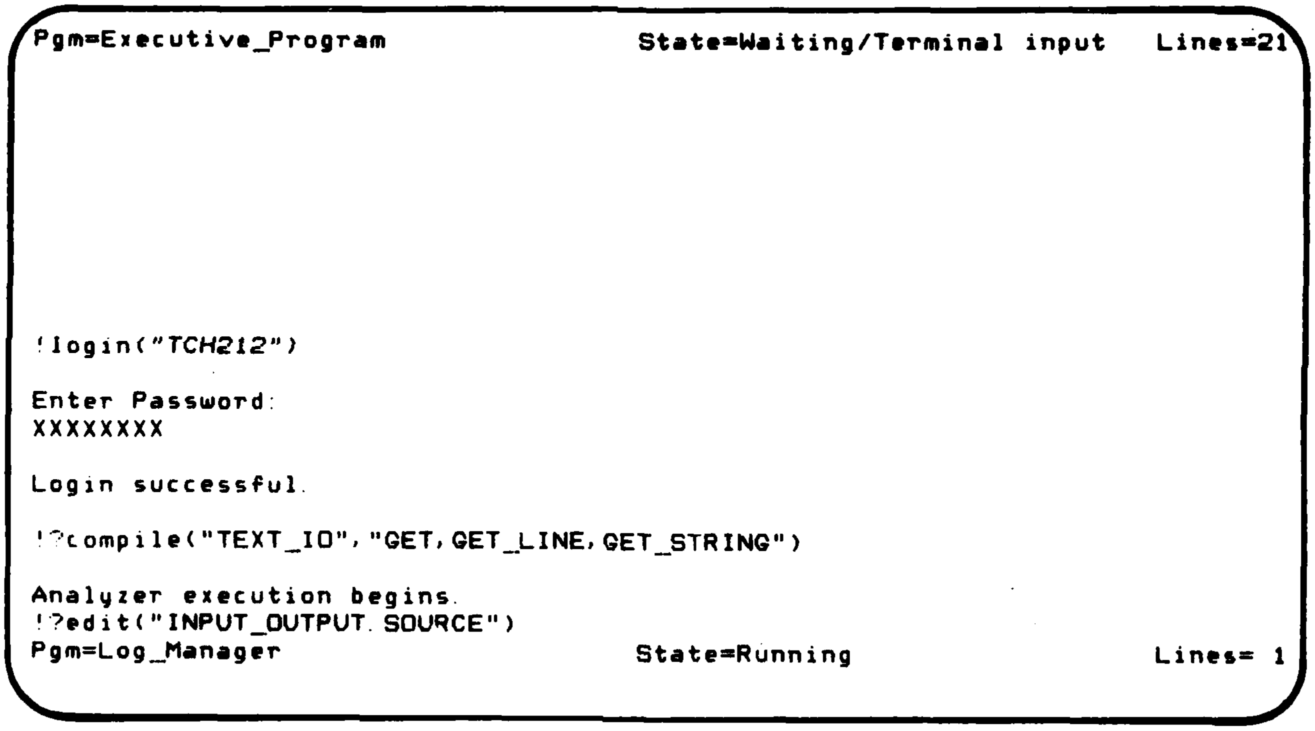


Figure E-9 Break Key Interrupts Compiler



```
Pgm=Executive_Program                    State=Waiting/Terminal input   Lines=21









!login("TCH212")

Enter Password:
XXXXXXXX

Login successful.

!?compile("TEXT_IO","GET,GET_LINE,GET_STRING")

Analyzer execution begins.
!?edit("INPUT_OUTPUT.SOURCE")
Pgm=Log_Manager                          State=Running                  Lines= 1
```

Figure E-10 User Activates Editor

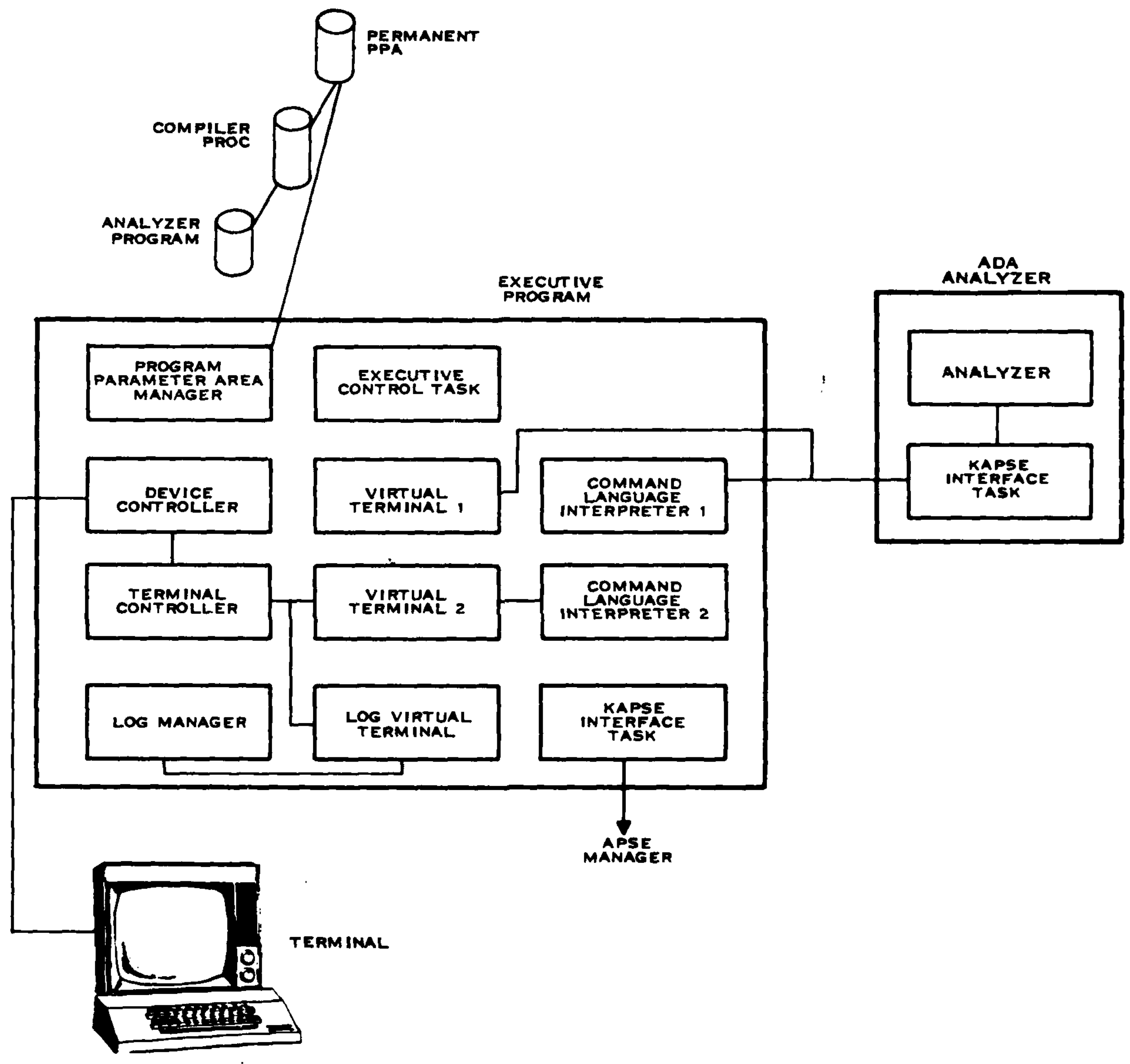


Figure E-11 Second CLI Activated

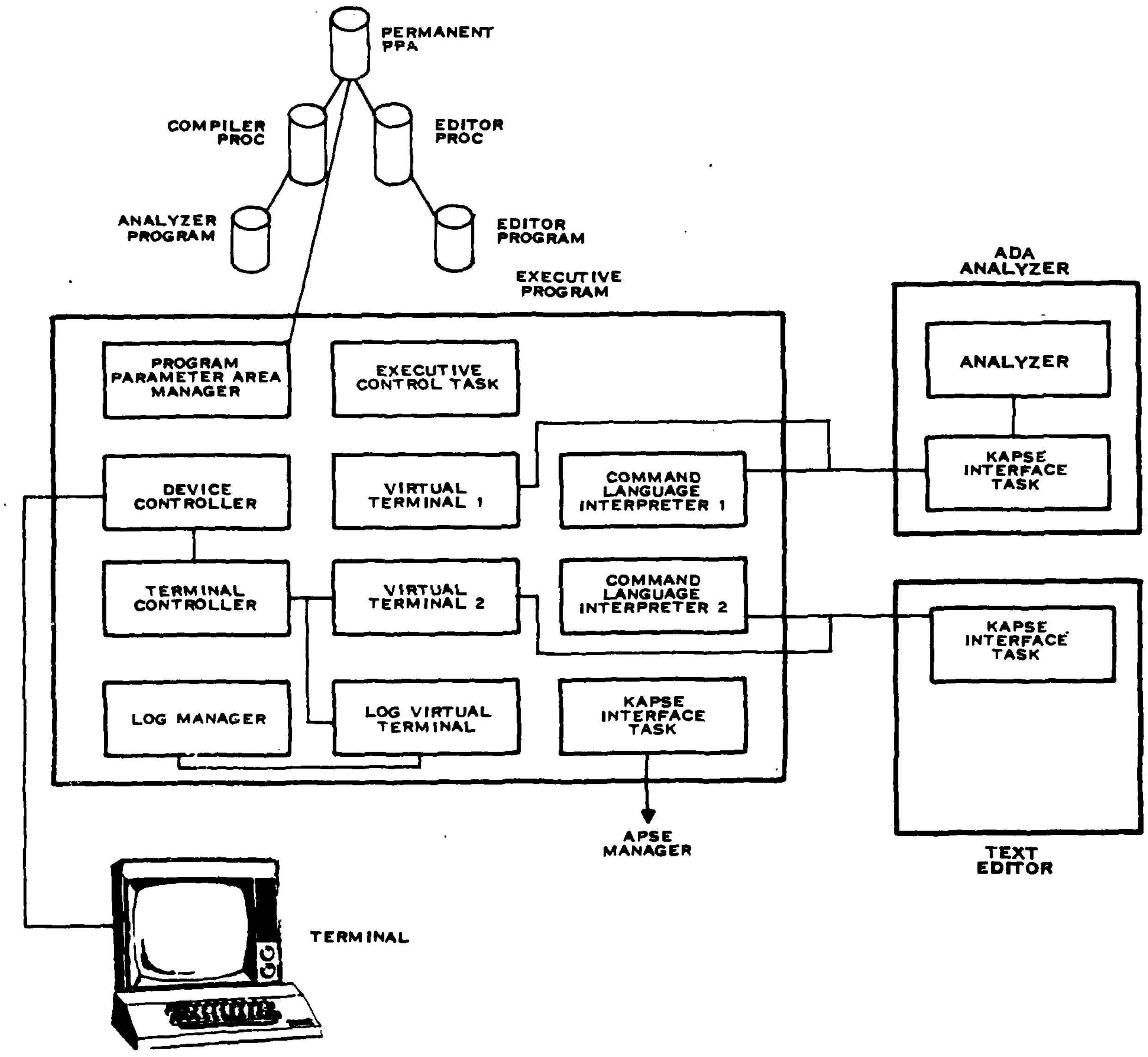


Figure E-12 Loading of Editor Begins

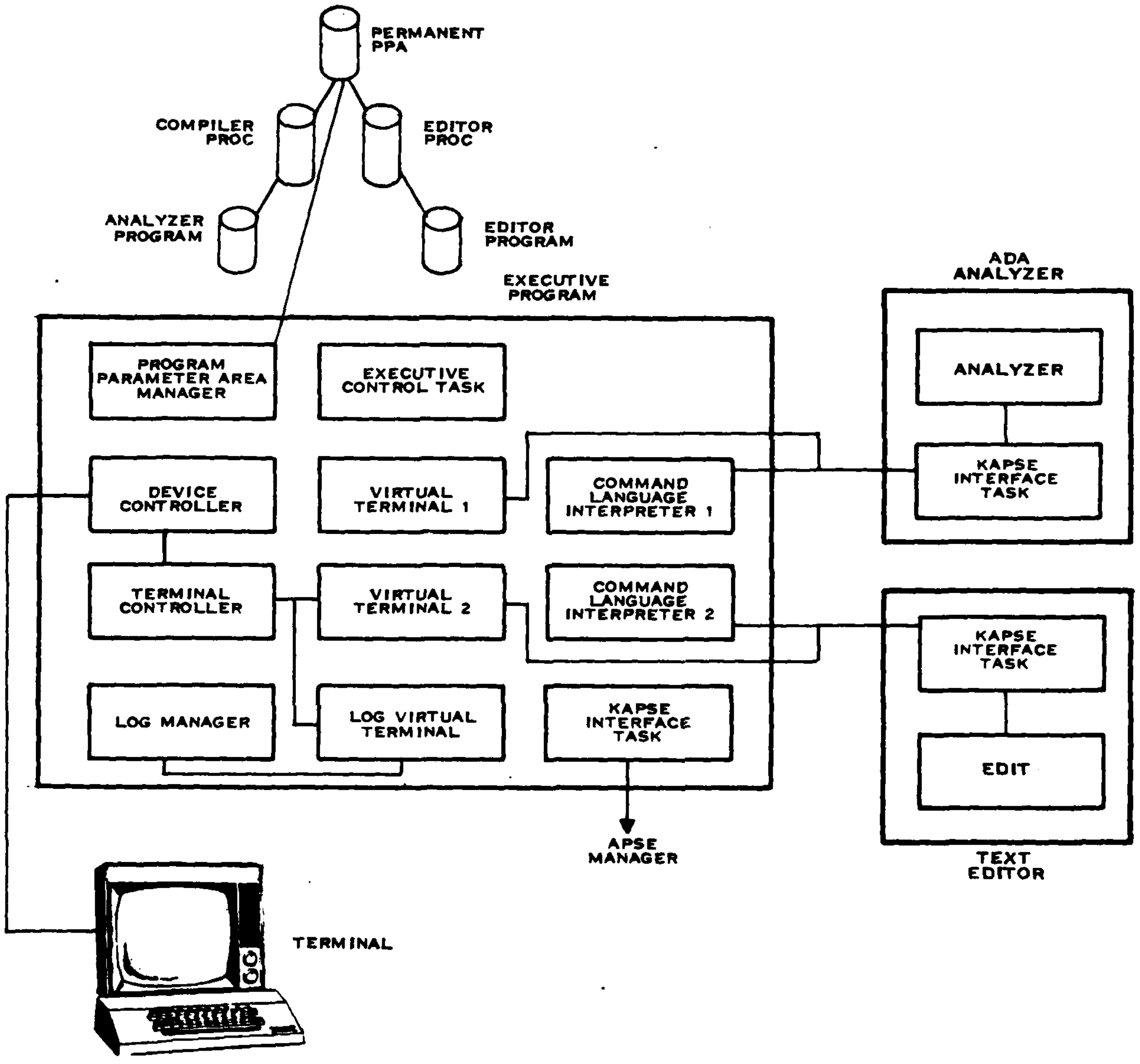
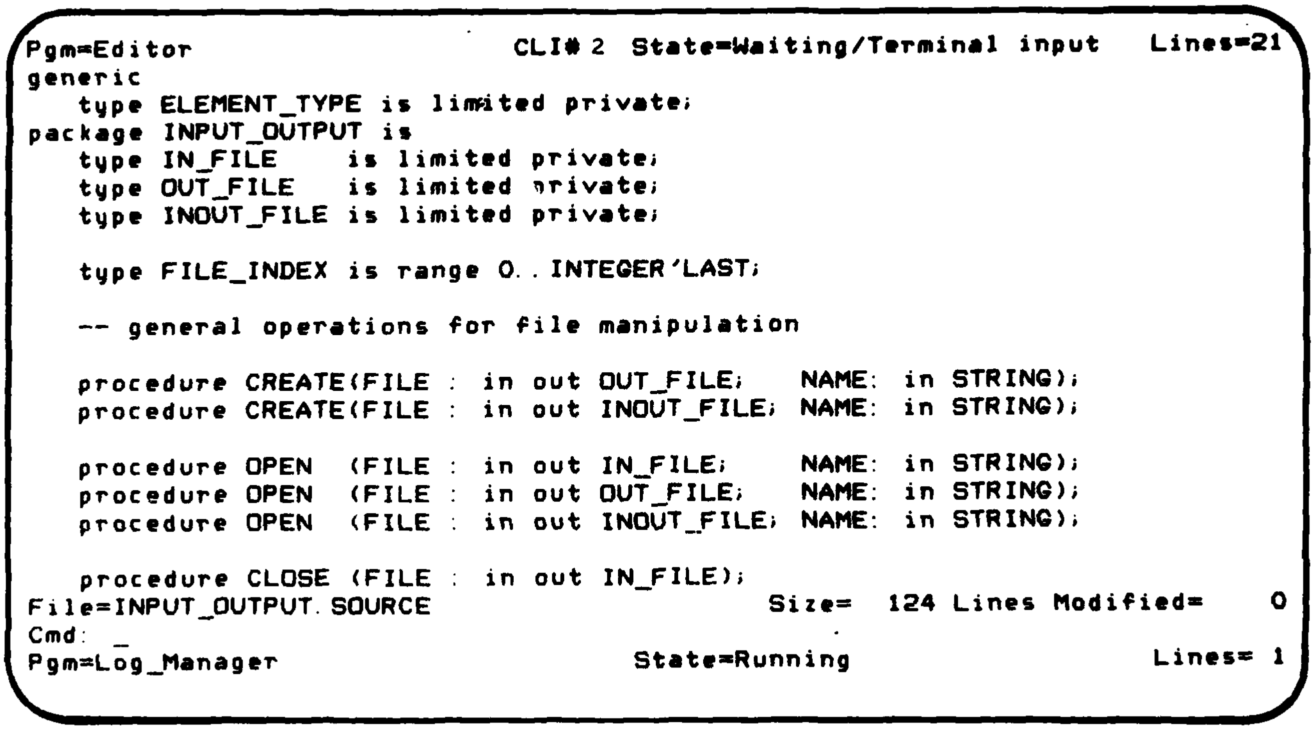
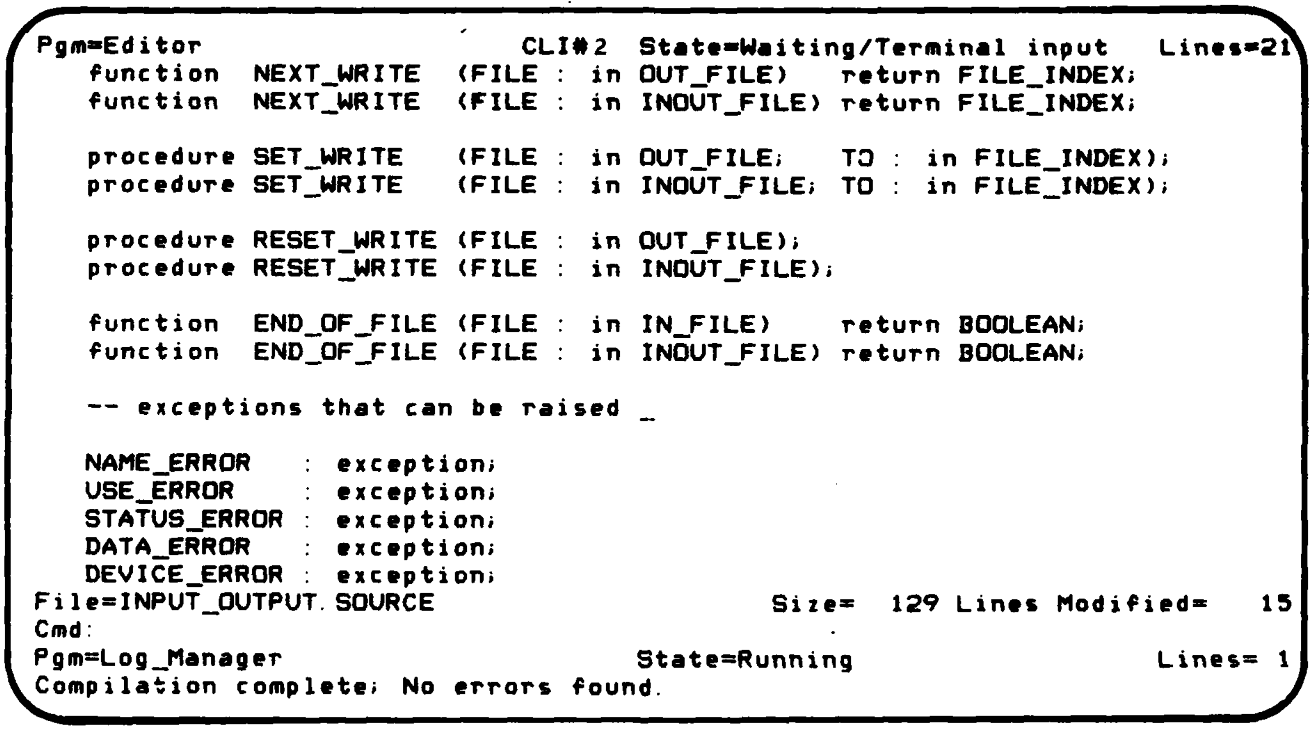


Figure E-13 Editor in Execution



```
Pgm=Editor                    CLI# 2  State=Waiting/Terminal input   Lines=21
generic
   type ELEMENT_TYPE is limited private;
package INPUT_OUTPUT is
   type IN_FILE    is limited private;
   type OUT_FILE   is limited private;
   type INOUT_FILE is limited private;

   type FILE_INDEX is range 0..INTEGER'LAST;

   -- general operations for file manipulation

   procedure CREATE(FILE : in out OUT_FILE;   NAME: in STRING);
   procedure CREATE(FILE : in out INOUT_FILE; NAME: in STRING);

   procedure OPEN  (FILE : in out IN_FILE;    NAME: in STRING);
   procedure OPEN  (FILE : in out OUT_FILE;   NAME: in STRING);
   procedure OPEN  (FILE : in out INOUT_FILE; NAME: in STRING);

   procedure CLOSE (FILE : in out IN_FILE);
File=INPUT_OUTPUT.SOURCE                        Size=  124 Lines Modified=    0
Cmd: _
Pgm=Log_Manager                        State=Running                   Lines= 1
```

Figure E-14 Edit Session Begins



```
Pgm=Editor                      CLI#2  State=Waiting/Terminal input   Lines=21
    function  NEXT_WRITE  (FILE : in OUT_FILE)    return FILE_INDEX;
    function  NEXT_WRITE  (FILE : in INOUT_FILE) return FILE_INDEX;

    procedure SET_WRITE   (FILE : in OUT_FILE;    TO : in FILE_INDEX);
    procedure SET_WRITE   (FILE : in INOUT_FILE; TO : in FILE_INDEX);

    procedure RESET_WRITE (FILE : in OUT_FILE);
    procedure RESET_WRITE (FILE : in INOUT_FILE);

    function  END_OF_FILE (FILE : in IN_FILE)     return BOOLEAN;
    function  END_OF_FILE (FILE : in INOUT_FILE) return BOOLEAN;

    -- exceptions that can be raised _

    NAME_ERROR   : exception;
    USE_ERROR    : exception;
    STATUS_ERROR : exception;
    DATA_ERROR   : exception;
    DEVICE_ERROR : exception;
File=INPUT_OUTPUT.SOURCE                         Size=  129 Lines Modified=   15
Cmd:
Pgm=Log_Manager                      State=Running                    Lines= 1
Compilation complete; No errors found.
```

Figure E-15 Compiler Sends Termination Message

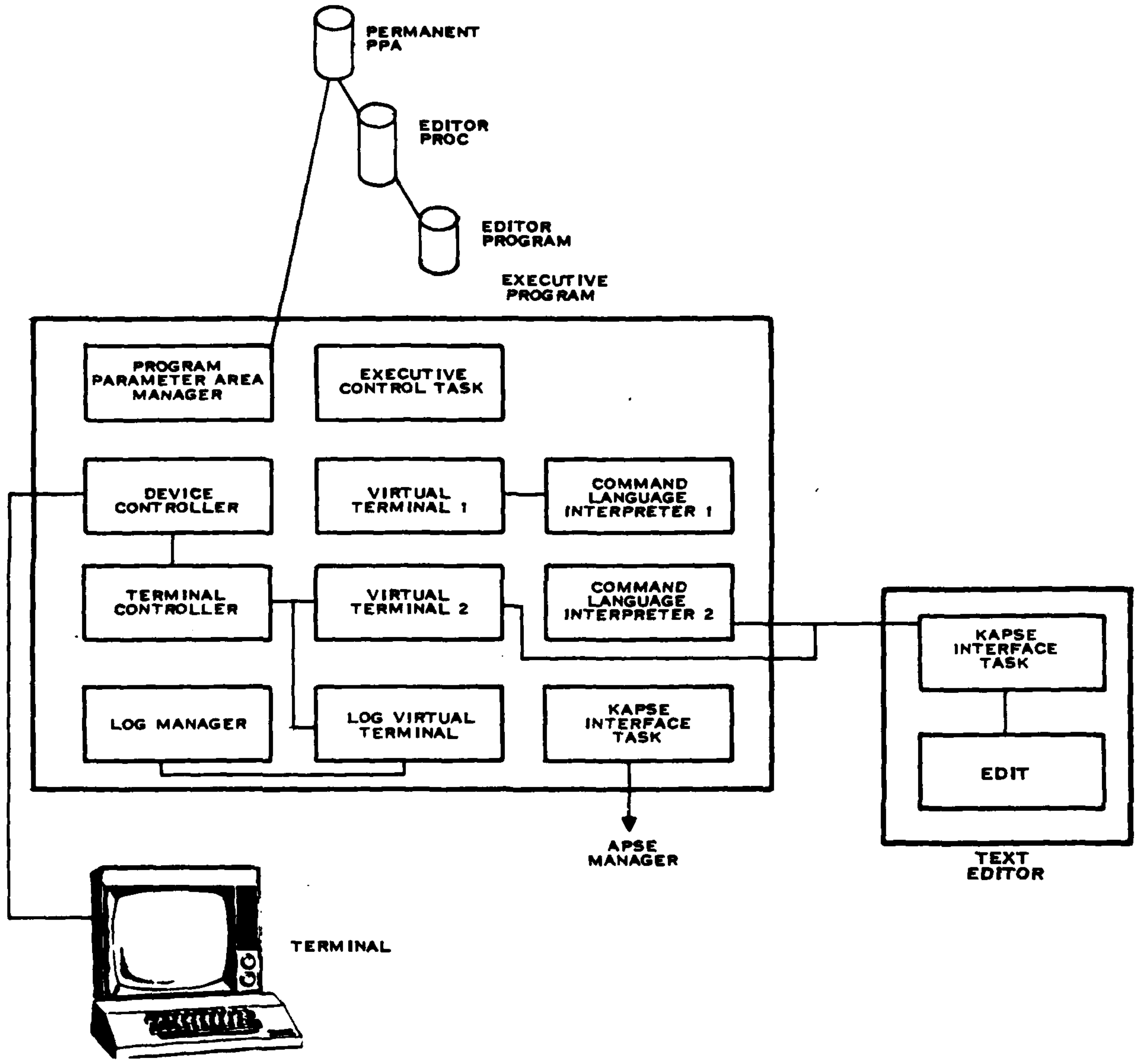


Figure E-16 Compiler Terminates

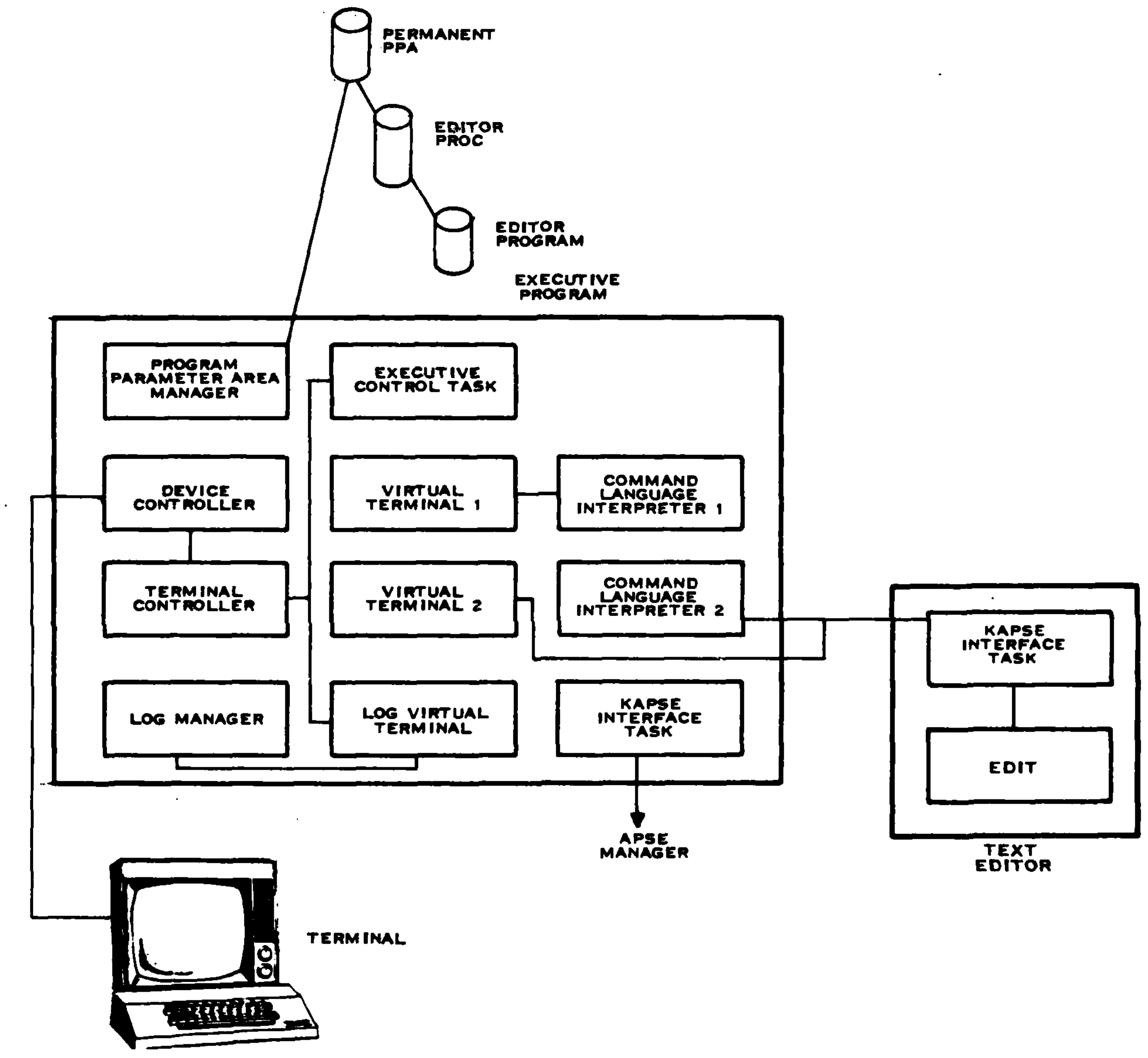
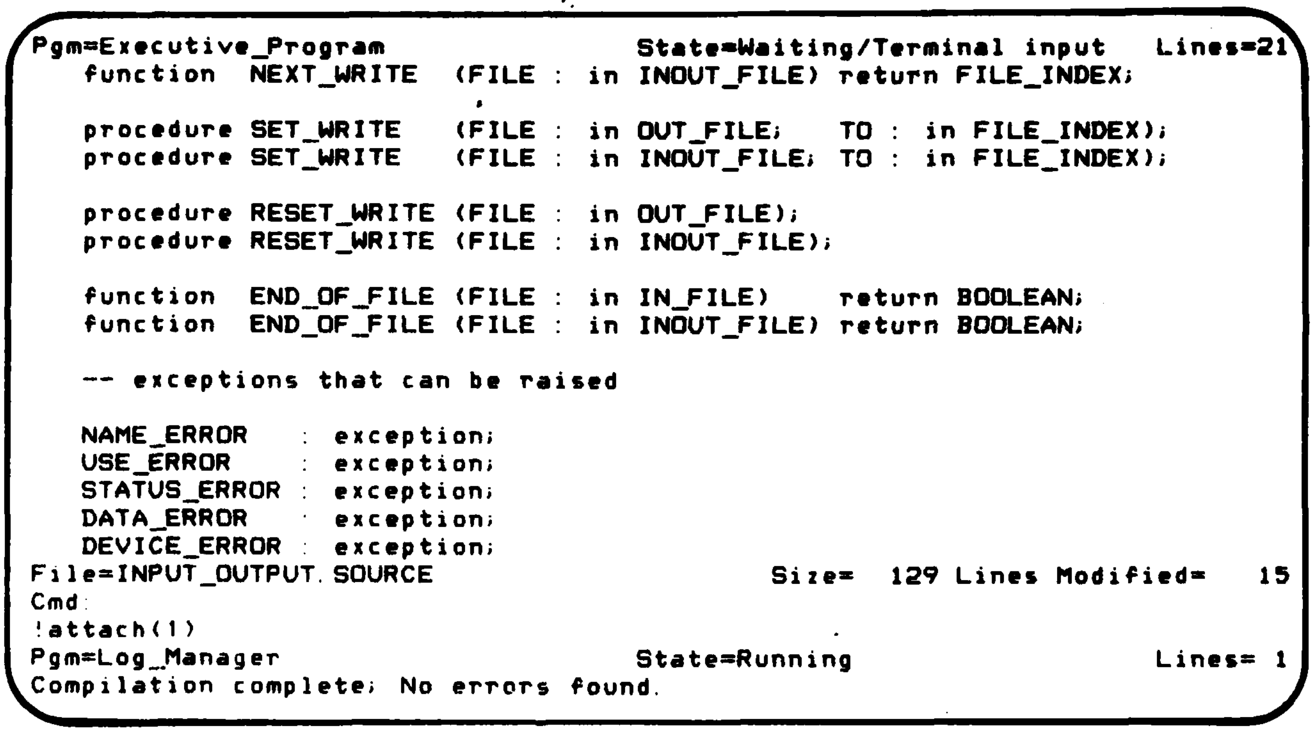


Figure E-17 Break Key Interrupts Editor



```
Pgm=Executive_Program                State=Waiting/Terminal input   Lines=21
   function  NEXT_WRITE  (FILE : in INOUT_FILE) return FILE_INDEX;
                                .
   procedure SET_WRITE   (FILE : in OUT_FILE;   TO : in FILE_INDEX);
   procedure SET_WRITE   (FILE : in INOUT_FILE; TO : in FILE_INDEX);

   procedure RESET_WRITE (FILE : in OUT_FILE);
   procedure RESET_WRITE (FILE : in INOUT_FILE);

   function  END_OF_FILE (FILE : in IN_FILE)    return BOOLEAN;
   function  END_OF_FILE (FILE : in INOUT_FILE) return BOOLEAN;

   -- exceptions that can be raised

   NAME_ERROR   : exception;
   USE_ERROR    : exception;
   STATUS_ERROR : exception;
   DATA_ERROR   : exception;
   DEVICE_ERROR : exception;
File=INPUT_OUTPUT.SOURCE                       Size=  129 Lines Modified=   15
Cmd:
!attach(1)
Pgm=Log_Manager                      State=Running                   Lines= 1
Compilation complete; No errors found.
```

Figure E-18 First CLI Resumed

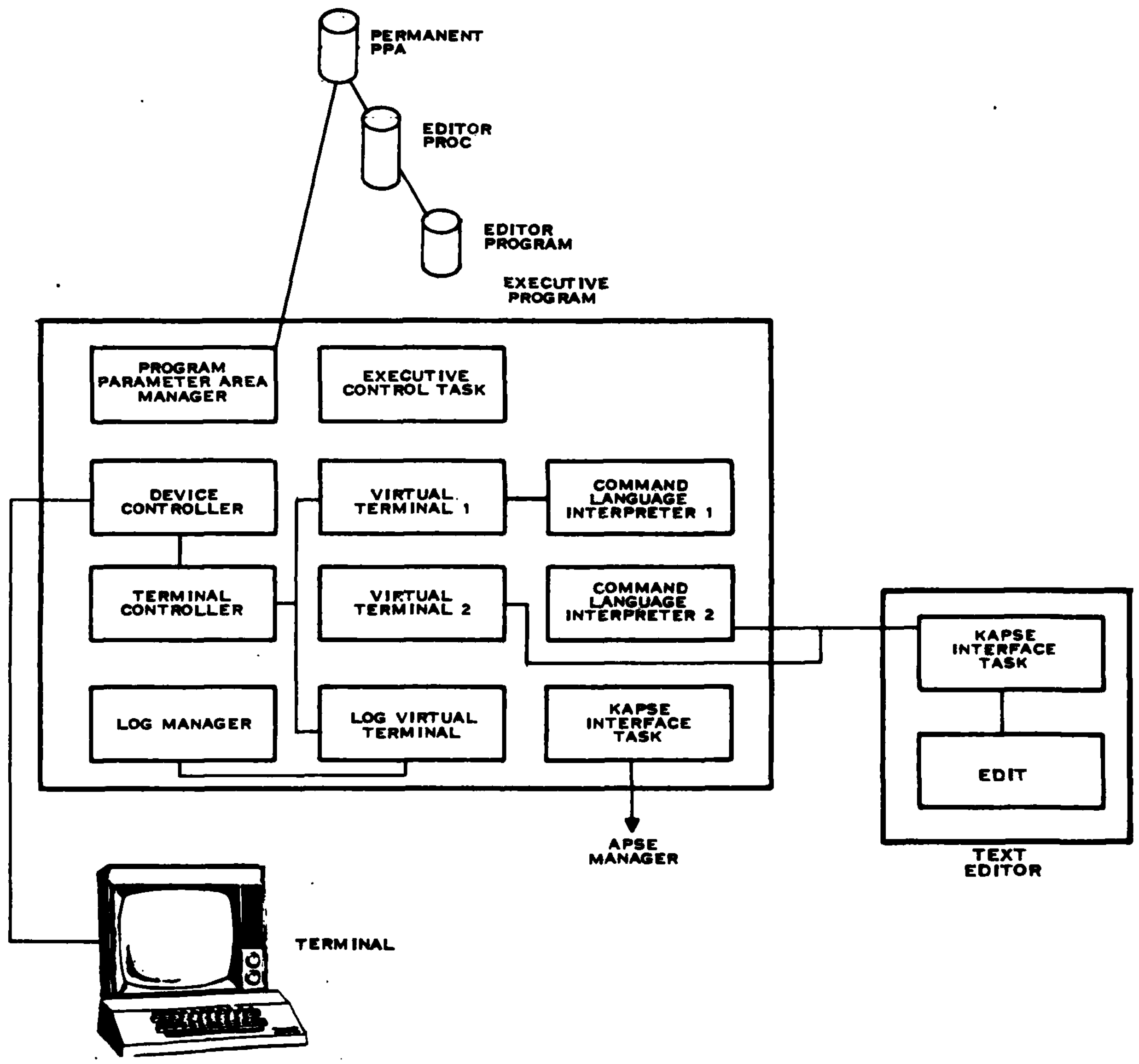
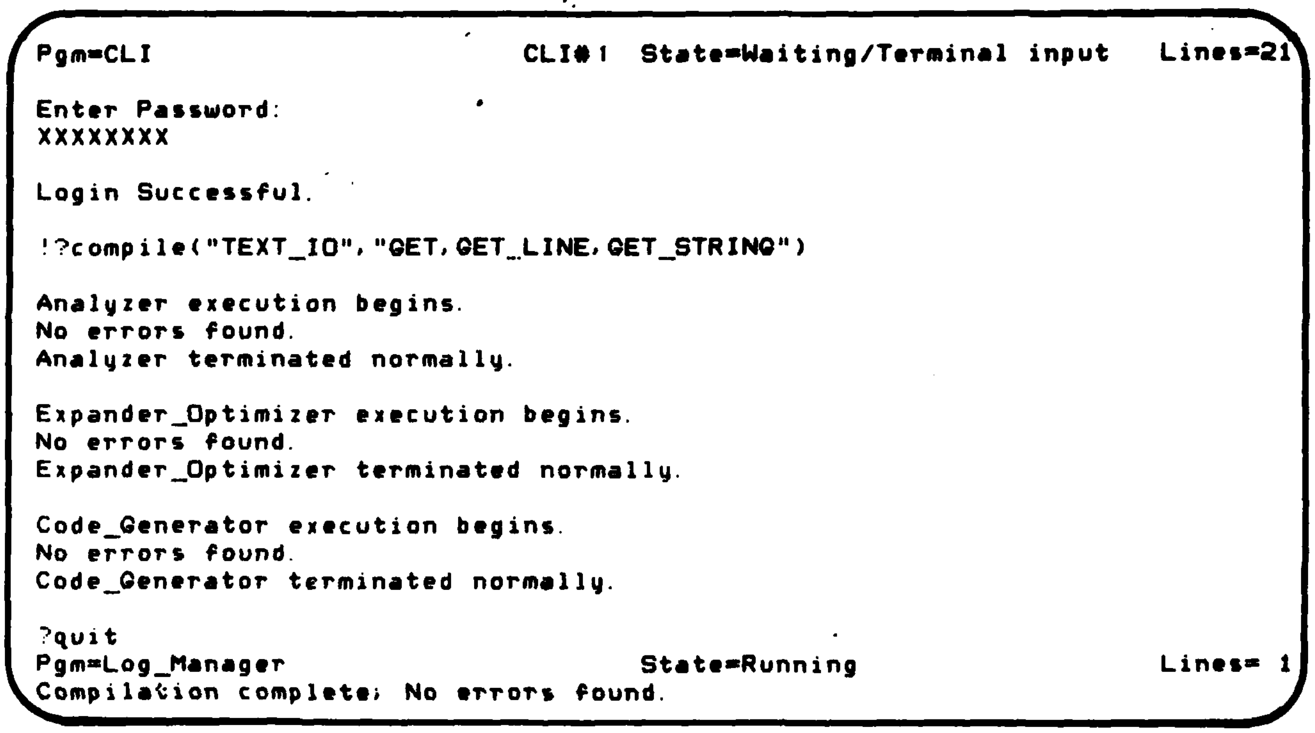


Figure E-19 First CLI in Execution



```
Pgm=CLI                    CLI#1  State=Waiting/Terminal input   Lines=21

Enter Password:
XXXXXXXX

Login Successful.

!?compile("TEXT_IO","GET,GET_LINE,GET_STRING")

Analyzer execution begins.
No errors found.
Analyzer terminated normally.

Expander_Optimizer execution begins.
No errors found.
Expander_Optimizer terminated normally.

Code_Generator execution begins.
No errors found.
Code_Generator terminated normally.

?quit
Pgm=Log_Manager                     State=Running                Lines= 1
Compilation complete; No errors found.
```

Figure E-20 User Terminates First CLI

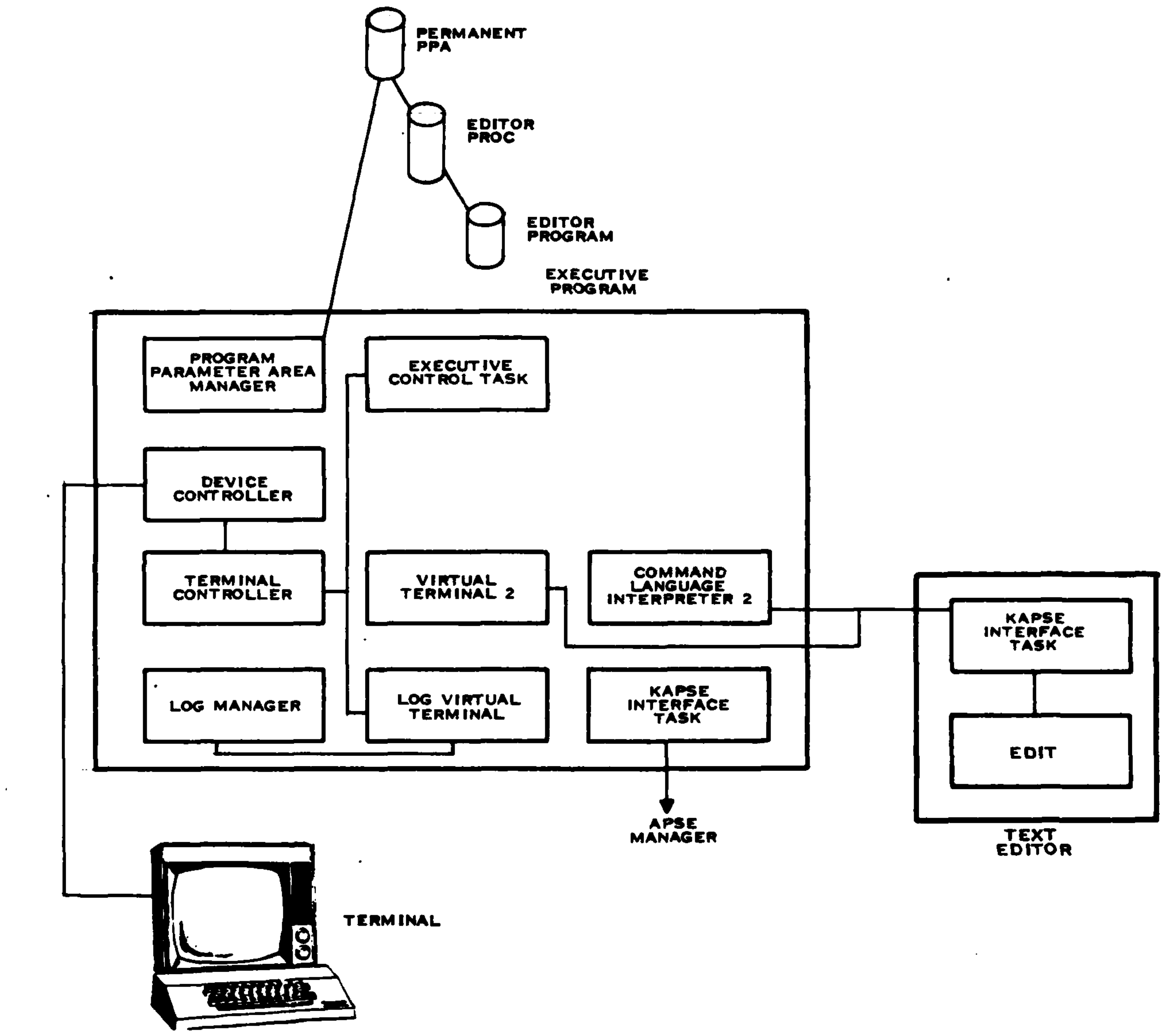
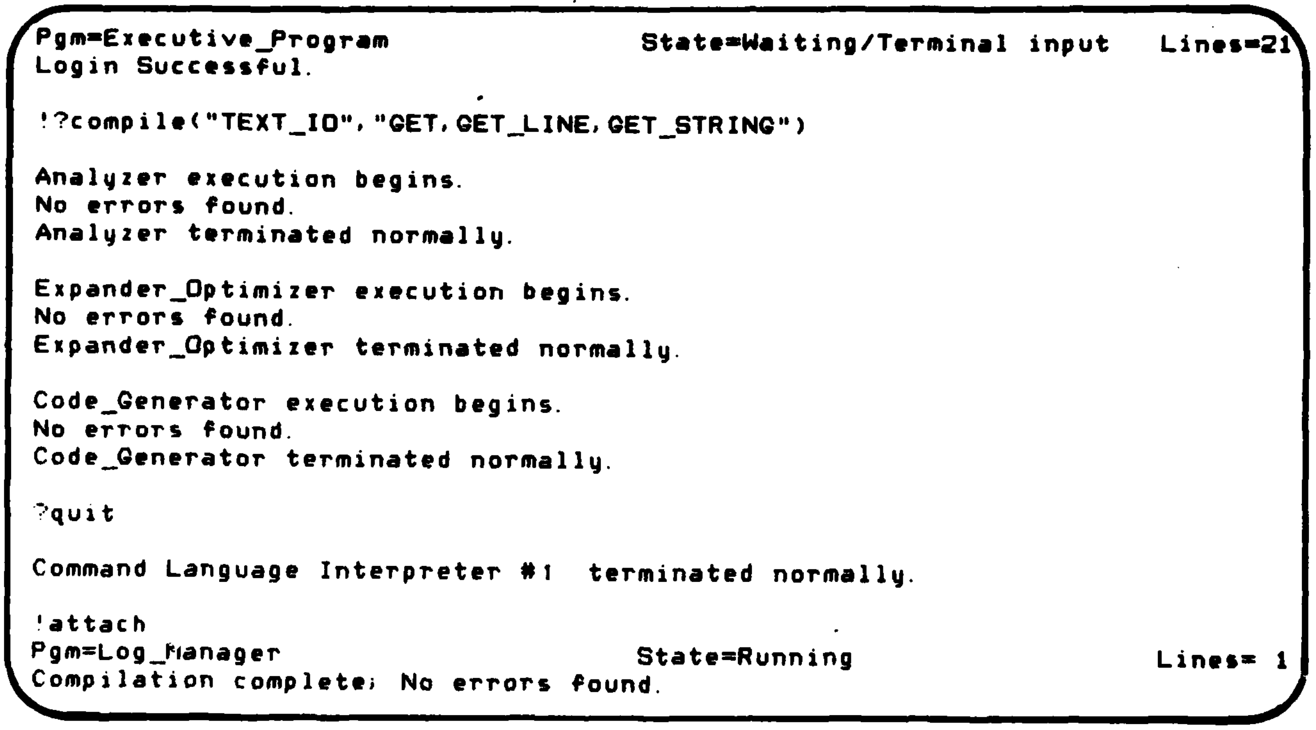


Figure E-21 First CLI Terminated



```
Pgm=Executive_Program                       State=Waiting/Terminal input   Lines=21
Login Successful.

!?compile("TEXT_IO","GET,GET_LINE,GET_STRING")

Analyzer execution begins.
No errors found.
Analyzer terminated normally.

Expander_Optimizer execution begins.
No errors found.
Expander_Optimizer terminated normally.

Code_Generator execution begins.
No errors found.
Code_Generator terminated normally.

?quit

Command Language Interpreter #1  terminated normally.

!attach
Pgm=Log_Manager                             State=Running                  Lines= 1
Compilation complete; No errors found.
```

Figure E-22 User Resumes Edit Session

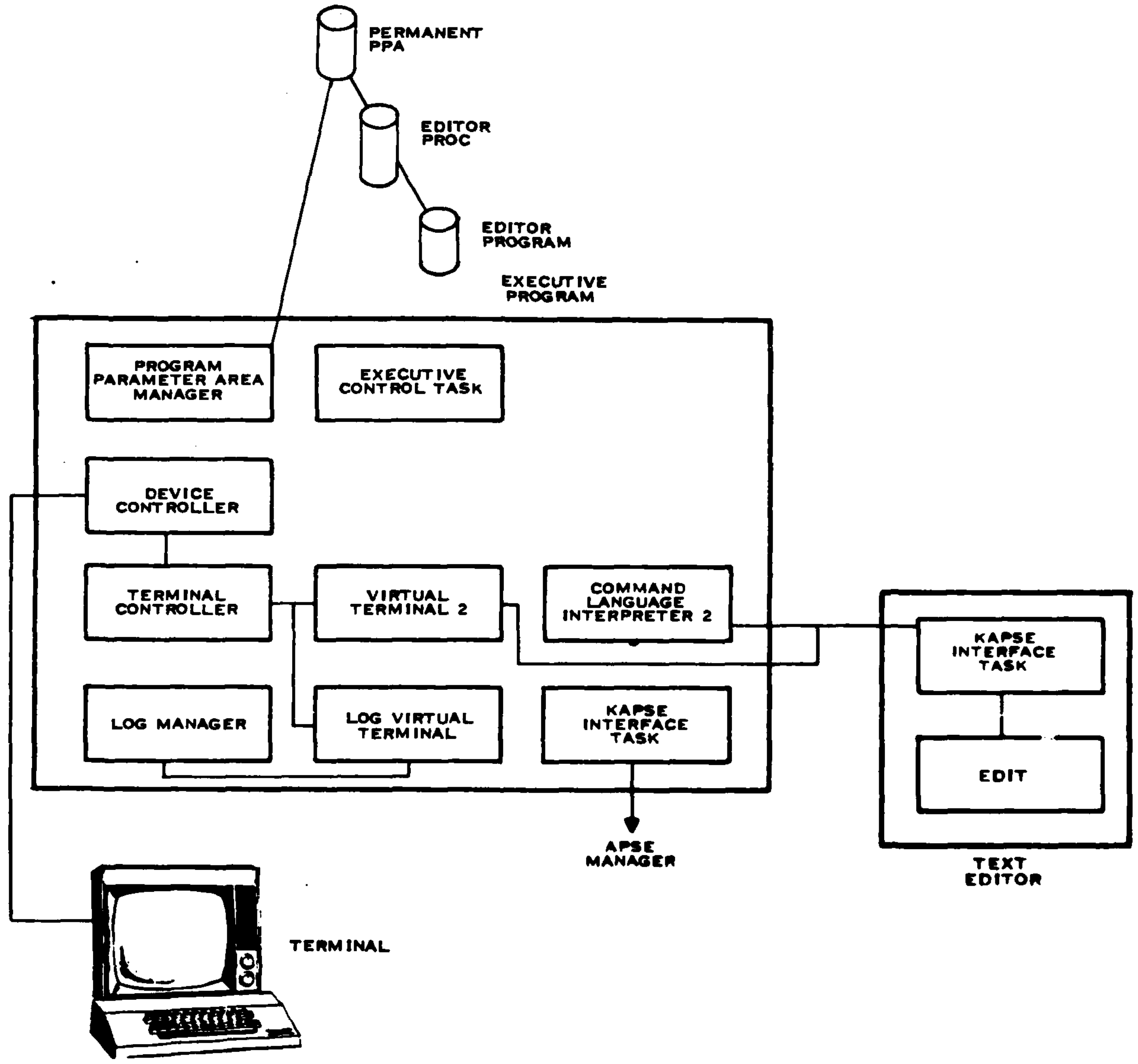
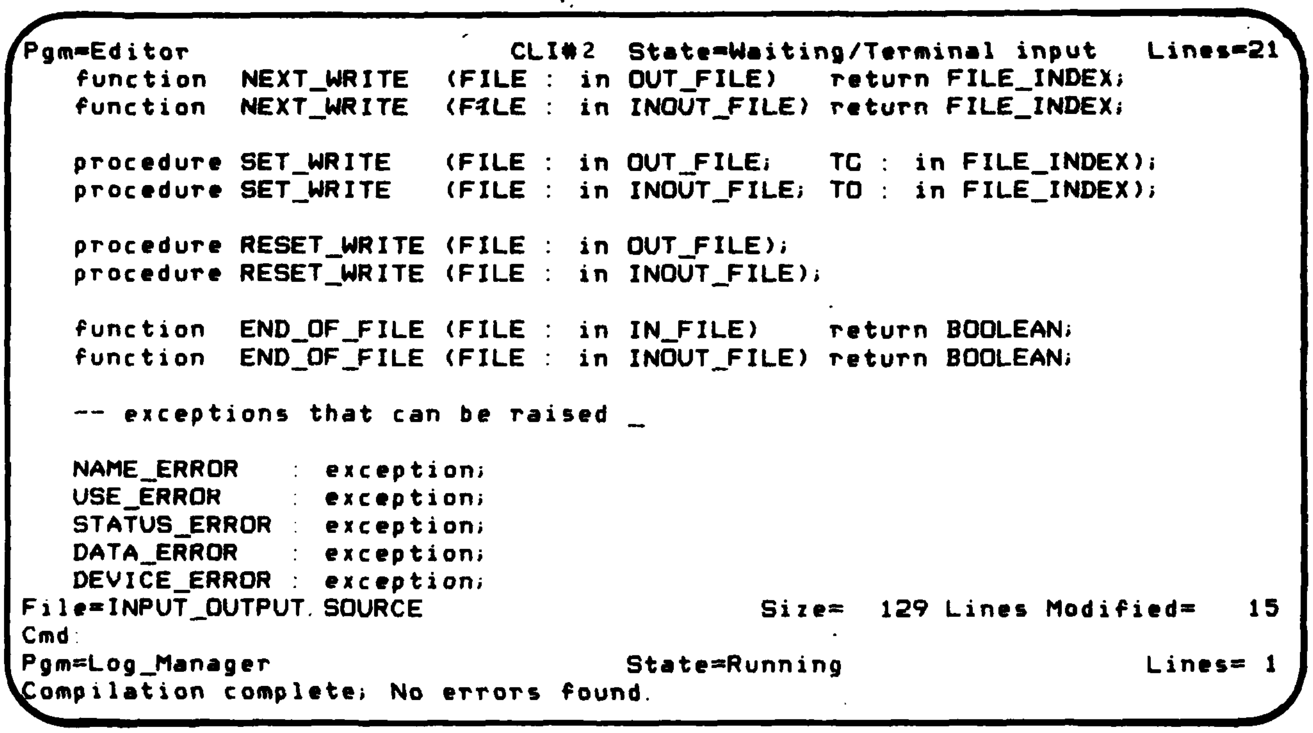
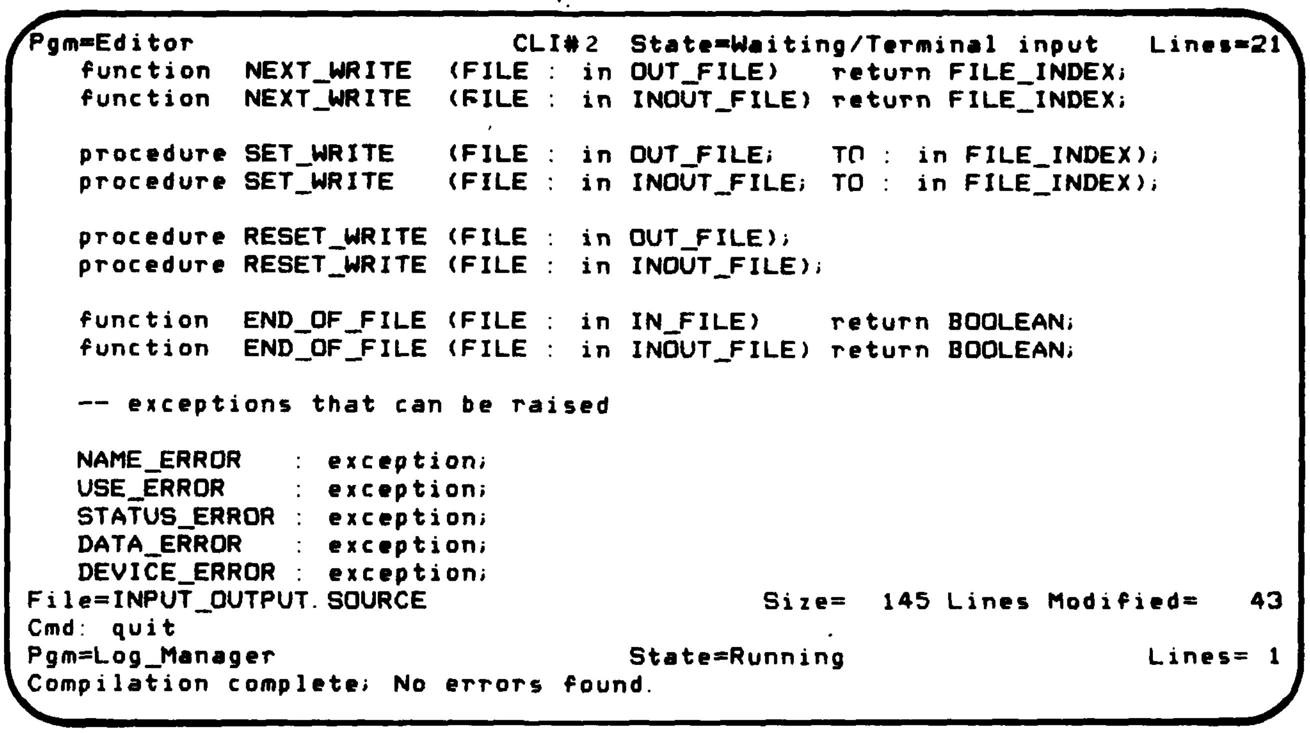


Figure E-23 Editor Resumed



```
Pgm=Editor                      CLI#2  State=Waiting/Terminal input   Lines=21
   function  NEXT_WRITE  (FILE : in OUT_FILE)   return FILE_INDEX;
   function  NEXT_WRITE  (FILE : in INOUT_FILE) return FILE_INDEX;

   procedure SET_WRITE   (FILE : in OUT_FILE;   TC : in FILE_INDEX);
   procedure SET_WRITE   (FILE : in INOUT_FILE; TO : in FILE_INDEX);

   procedure RESET_WRITE (FILE : in OUT_FILE);
   procedure RESET_WRITE (FILE : in INOUT_FILE);

   function  END_OF_FILE (FILE : in IN_FILE)    return BOOLEAN;
   function  END_OF_FILE (FILE : in INOUT_FILE) return BOOLEAN;

   -- exceptions that can be raised _

   NAME_ERROR   : exception;
   USE_ERROR    : exception;
   STATUS_ERROR : exception;
   DATA_ERROR   : exception;
   DEVICE_ERROR : exception;
File=INPUT_OUTPUT.SOURCE                    Size=  129 Lines Modified=   15
Cmd:
Pgm=Log_Manager                    State=Running                     Lines= 1
Compilation complete; No errors found.
```

Figure E-24 Editor Resumes at Point of Suspension



```
Pgm=Editor                    CLI#2  State=Waiting/Terminal input   Lines=21
   function  NEXT_WRITE  (FILE : in OUT_FILE)    return FILE_INDEX;
   function  NEXT_WRITE  (FILE : in INOUT_FILE) return FILE_INDEX;

   procedure SET_WRITE   (FILE : in OUT_FILE;    TO : in FILE_INDEX);
   procedure SET_WRITE   (FILE : in INOUT_FILE; TO : in FILE_INDEX);

   procedure RESET_WRITE (FILE : in OUT_FILE);
   procedure RESET_WRITE (FILE : in INOUT_FILE);

   function  END_OF_FILE (FILE : in IN_FILE)     return BOOLEAN;
   function  END_OF_FILE (FILE : in INOUT_FILE) return BOOLEAN;

   -- exceptions that can be raised

   NAME_ERROR   : exception;
   USE_ERROR    : exception;
   STATUS_ERROR : exception;
   DATA_ERROR   : exception;
   DEVICE_ERROR : exception;
File=INPUT_OUTPUT.SOURCE                      Size=  145 Lines Modified=   43
Cmd: quit
Pgm=Log_Manager                      State=Running                  Lines= 1
Compilation complete; No errors found.
```

Figure E-25 User Terminates Editor

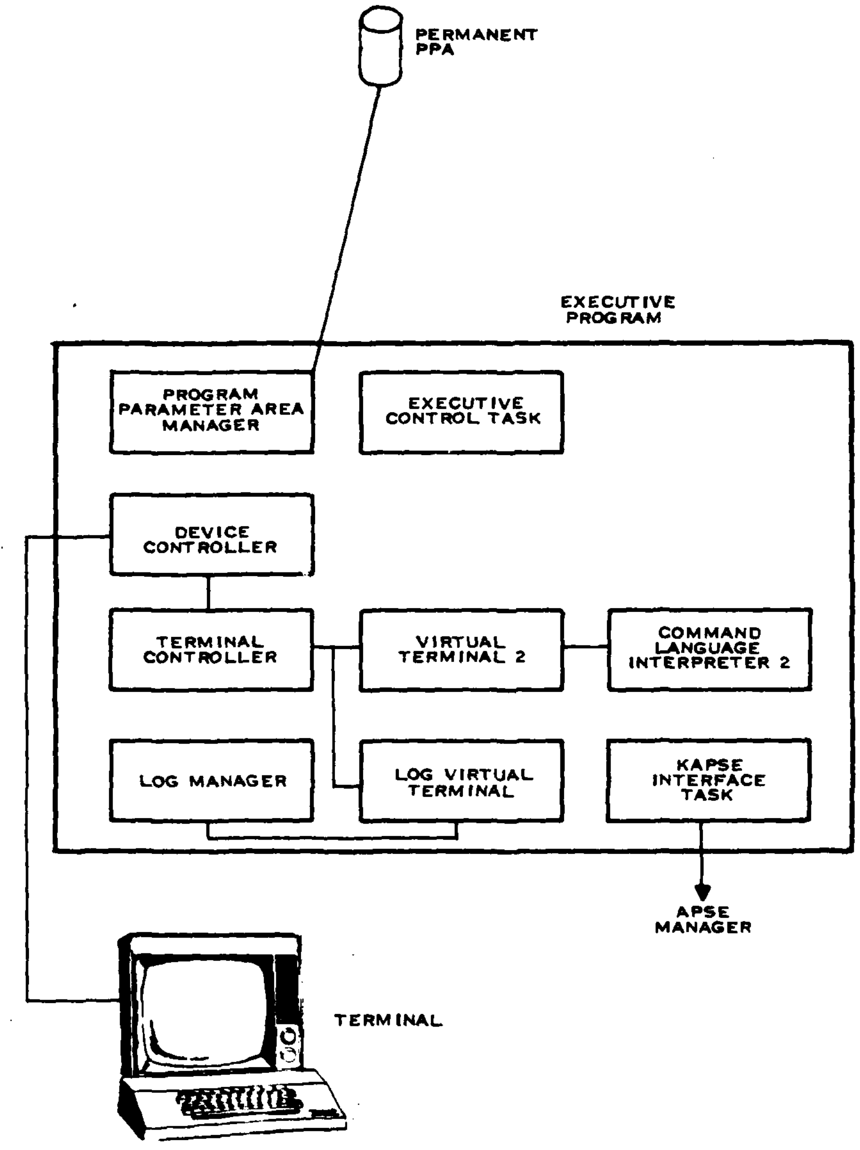


Figure E-26 Editor Terminates

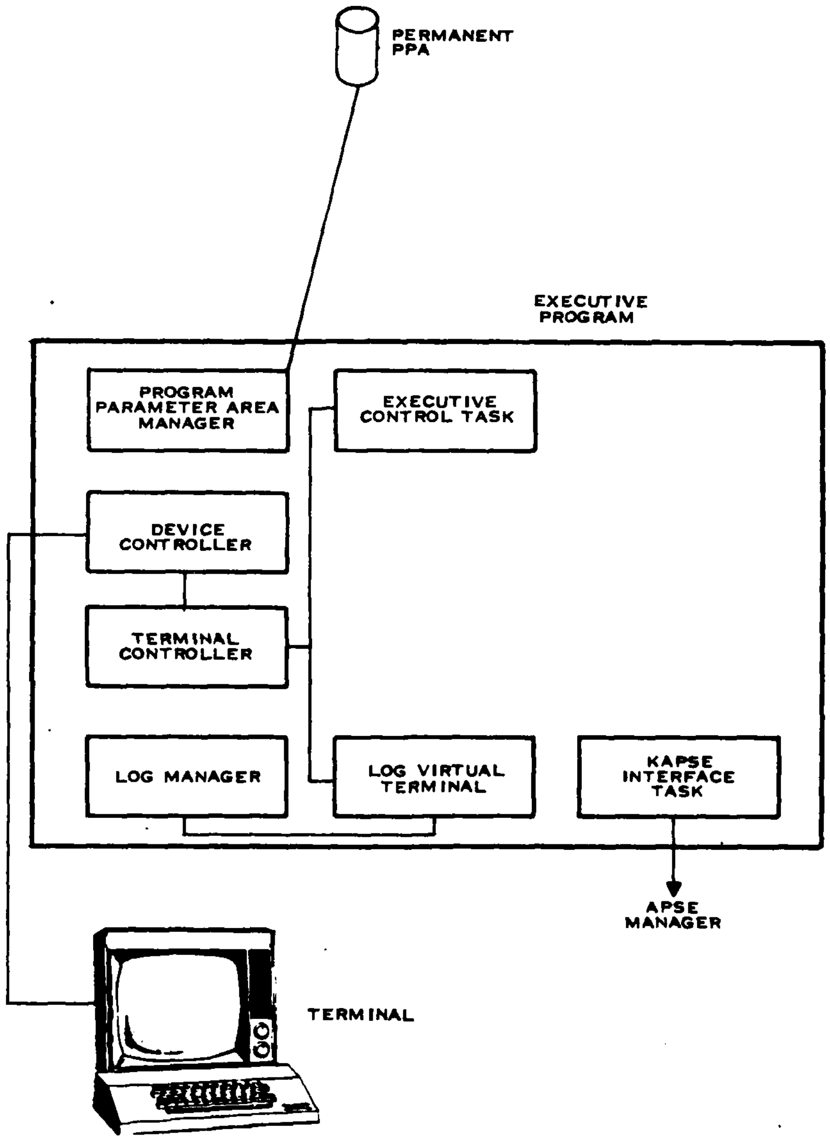
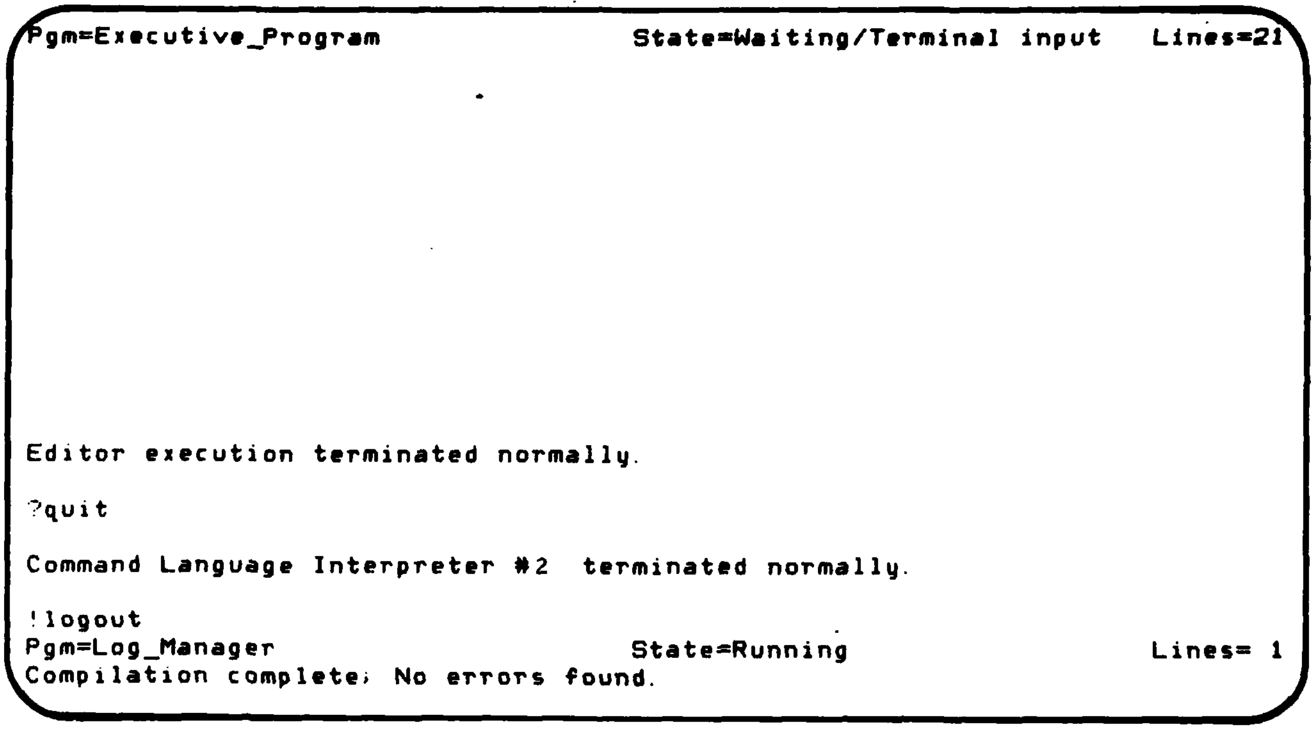


Figure E-27 Second CLI Terminates



```
Pgm=Executive_Program                   State=Waiting/Terminal input   Lines=21
















Editor execution terminated normally.

?quit

Command Language Interpreter #2  terminated normally.

!logout
Pgm=Log_Manager                         State=Running                  Lines= 1
Compilation complete; No errors found.
```

Figure E-28 User Enters LOGOUT Command

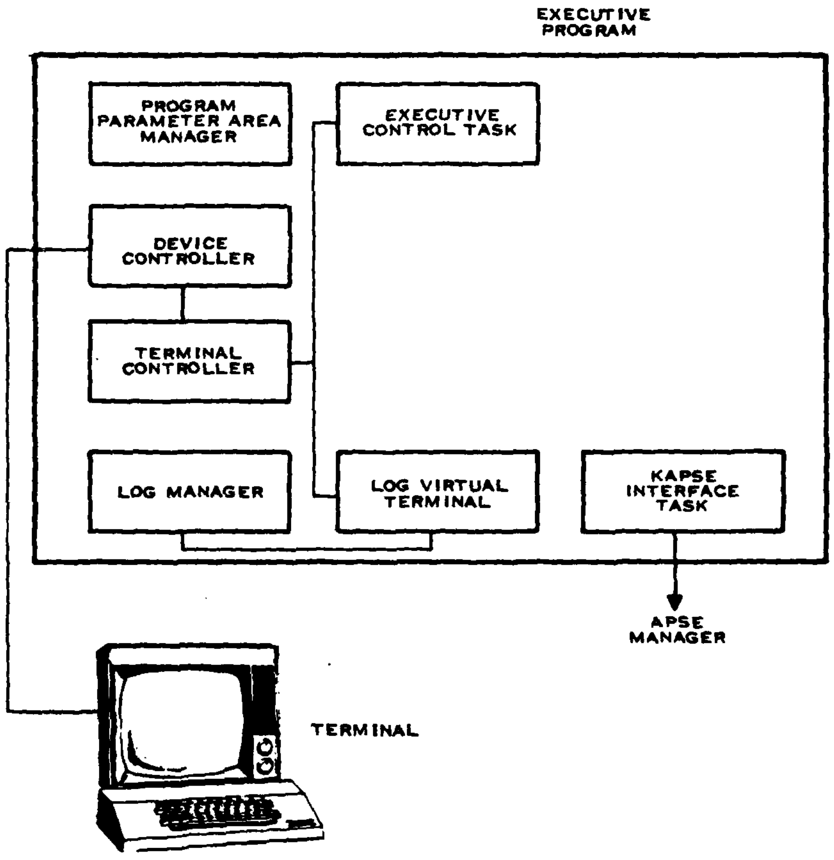


Figure E-29 LOGOUT Complete

APPENDIX F

REFERENCES

[BAK80] Baker, Henry G. Jr., List processing in real time on a serial computer, CACM 21, 4 (April 1978), 280-294.

[BAT79] Bate, Roger R., and Douglas S. Johnson, Putting Pascal to Work, Electronics, 7 (June 1979), 111-121.

[DIJ68] Dijkstra, E. W., Co-operating Sequential Process, in Programming Languages (ed. F. Genuys), Academic Press, New York, (1968).

[EVA80] Evans, Arthur Jr., Slides on tasking in Ada, Ada Implementors Newsletter, (September 1980).

[EVE80] Eventoff, W., D. Harvey, and R. J. Rice, The Rendezvous and Monitor Concepts: Is There an Efficiency Difference? SIGPLAN Notices 15, 11 (November 1980), 156-165.

[GRI71] Gries, D., Compiler Construction for Digital Computers, John Wiley and Sons, Inc., New York, (1971).

[HAB80] Habermann, A. N., and Isaac R. Nassi, Efficient implementation of Ada tasks, Carnegie-Mellon University Computer Science Department, (January 1980).

[HAL80] Hall, Dennis E., Deborah K. Scherrer, and Joseph S. Sventek, A Virtual Operating System, CACM, 23, 9 (September 1980), 495-502.

[IBM76] International Business Machines Corporation, IBM System/370 Principles of Operation, GA22-7000-5, (1976).

[IBM77] International Business Machines Corporation, IBM Virtual Machine Facility/370: Operating Systems in a Virtual Machine, GC20-1821-2, (1977).

[IBM79A] International Business Machines Corporation, IBM Virtual Machine Facility/370: CMS User's Guide, GC20-1819-2, (1979).

[IBM79B] International Business Machines Corporation, IBM Virtual Machine Facility/370: System Programmer's Guide, GC20-1807-7, (1979).

[IBM80] International Business Machines Corporation, IBM Virtual Machine/System Product: System Programmer's Guide, SC19-6203-0, (1980).

[JEN79] Jensen, R. M., A formal approach for communication between logically isolated virtual machines, IBM Systems Journal, 18, 1 (January 1979), 71-92.

[MAC79] MacKinnon, R. A., The changing virtual machine environment: Interfaces to real hardware, virtual hardware, and other virtual machines, IBM Systems Journal, 18, 1 (January 1979), 18-46.

[PER78] Perkin-Elmer Corporation, M83-Series Models 8/32, 8/32C, and 8/32D Processors User Manual, Publication Number 29-428R06, Computer Systems Division, Perkin-Elmer Corporation, Oceanport, N. J., (1978).

[PER79] Perkin-Elmer Corporation, OS/32 Programmer Reference Manual, Publication Number S29-613R04, Computer Systems Division, Perkin-Elmer Corporation, Oceanport, N. J., (1979).

[SEA79] Seawright, L. H., and R. A. MacKinnon, VM/370 -- a study of multiplicity and usefulness, IBM Systems Journal, 18, 1 (January 1979), 4-17.

[SCH79] Scherrer, Deborah K., Dennis E. Hall, and Joseph S. Sventek, Cookbook: Instructions for Implementing the LBL Software Tools Package, Internal Report LBID 098 (Version 2), Lawrence Berkeley Laboratory, University of California, Berkeley, California, (1979).

[STE75] Steele, Guy L. Jr. Multiprocessing compactifying garbage collection, CACM 18, 9 (September 1975), 495-508.

[TI81A] Texas Intruments Incorporated, Design of the Ada Integrated Environment, Equipment Group, Texas Instruments Incorporated, Lewisville, Texas, (1981).

[TI81B] Texas Intruments Incorporated, Device Independent File I/O User's Manual, MP386, Semiconductor Group, Texas Instruments Incorporated, Houston, Texas, (1981).

[TI81C] Texas Intruments Incorporated, Microprocessor Pascal Executive User's Manual, MP385, Semiconductor Group, Texas Instruments Incorporated, Houston, Texas, (1981).

[TI81D] Texas Intruments Incorporated, Microprocessor Pascal System User's Manual, MP351 (Revision B), Semiconductor Group, Texas Instruments Incorporated, Houston, Texas, (1981).

[WAD76] Wadler, Philip L., Analysis of an algortihm for real-time garbage collection, CACM 19, 9 (September 1976), 491-500.

[WAR79] Warren, Scott K., and Dennis Abbe, Rosetta Smalltalk: A Conversational, Extensible Microcomputer Language, SIGSMALL Newsletter, Volume 5, Number 2 (April 1979), 36-45.

FILMED

8